Sentinel Scheduling: A Model for Compiler-Controlled Speculative Execution

Scott A. Mahlke William Y. Chen Roger A. Bringmann Richard E. Hank Wen-mei W. Hwu
University of Illinois, Urbana-Champaign, IL

B. Ramakrishna Rau Michael S. Schlansker
Hewlett Packard Laboratories, Palo Alto, CA *

Abstract

Speculative execution is an important source of parallelism for VLIW and superscalar processors. A serious challenge with compiler-controlled speculative execution is to efficiently handle exceptions for speculative instructions. In this paper, a set of architectural features and compile-time scheduling support collectively referred to as *sentinel scheduling* is introduced. Sentinel scheduling provides an effective framework for both compiler-controlled speculative execution and exception handling. All program exceptions are accurately detected and reported in a timely manner with sentinel scheduling. Recovery from exceptions is also ensured with the model. Experimental results show the effectiveness of sentinel scheduling for exploiting instruction-level parallelism and the overhead associated with exception handling.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles–*associative memories*; C.0 [**Computer Systems Organization**]: General–*hardware/software interfaces; instruction set design; system architectures*; C.1.2 [**Processor Architectures**]: Single Data Stream Architectures–*pipeline processors*; D.2.4 [**Software Engineering**]: Testing and Debugging–*error handling and recovery*; D.3.3 [**Programming Languages**]: Processors–*code generation; compilers; optimization*;

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: exception detection, exception recovery, instruction-level parallelism, instruction scheduling, speculative execution, superscalar processor, VLIW processor

*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. Scott Mahlke is supported by a fellowship provided by Intel Foundation. Authors' addresses: S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, and W. W. Hwu are with the Center for Reliable and High-Performance Computing, University of Illinois, Urbana-Champaign, IL 61801. B. R. Rau and M. S. Schlansker are with Hewlett Packard Laboratories, Palo Alto, CA 94303.

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 results of branch conditions may be determined early, scheduling techniques such as software pipelining [18] [16] [2] are effective for exposing ILP. Also, predicated instructions can be used in conjunction with software pipeline loop scheduling [19] or straight-line code scheduling [12] to mask out the effects of unnecessary instructions from alternate paths of control. For applications in which results of branch conditions may not be determined early, speculative execution of instructions is an important source of ILP [22] [23] [4].

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. A compiler may utilize speculative code motion to achieve higher performance in three major ways. First, in regions of the program where insufficient ILP exists to fully utilize the processor resources, useful instructions may be executed. Second, instructions starting long dependence chains may be executed early to reduce the length of critical paths. Finally, long latency instructions may be initiated early to overlap their execution with useful computation.

There are two problems though associated with speculative code motion. The first problem is that the result value of a speculative instruction that is not required to execute must not affect the execution of the subsequent instructions. This may be effectively achieved by compile-time renaming transformations. A more serious problem with speculative code motion is correctly handling

exceptions. An exception that occurs for a speculative instruction which is not supposed to execute must be ignored. On the other hand, an exception for a speculative instruction that is supposed to execute must be signaled. Accurately detecting and reporting exceptions are required to identify program execution errors at the time of occurrence. Also, for exceptions which do not terminate program execution, exception recovery must 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 efficiently handle exceptions that occur for speculative 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 [8], modulo scheduling [18], and enhanced pipelining [6] may effectively utilize the speculative execution models discussed in this paper. Tirumalai *et al.* showed that modulo scheduling of while loops depends on speculative support to achieve high performance [23]. Without speculative support, dependences limit the amount of execution overlap between loop iterations.

2.1 Superblock Structure

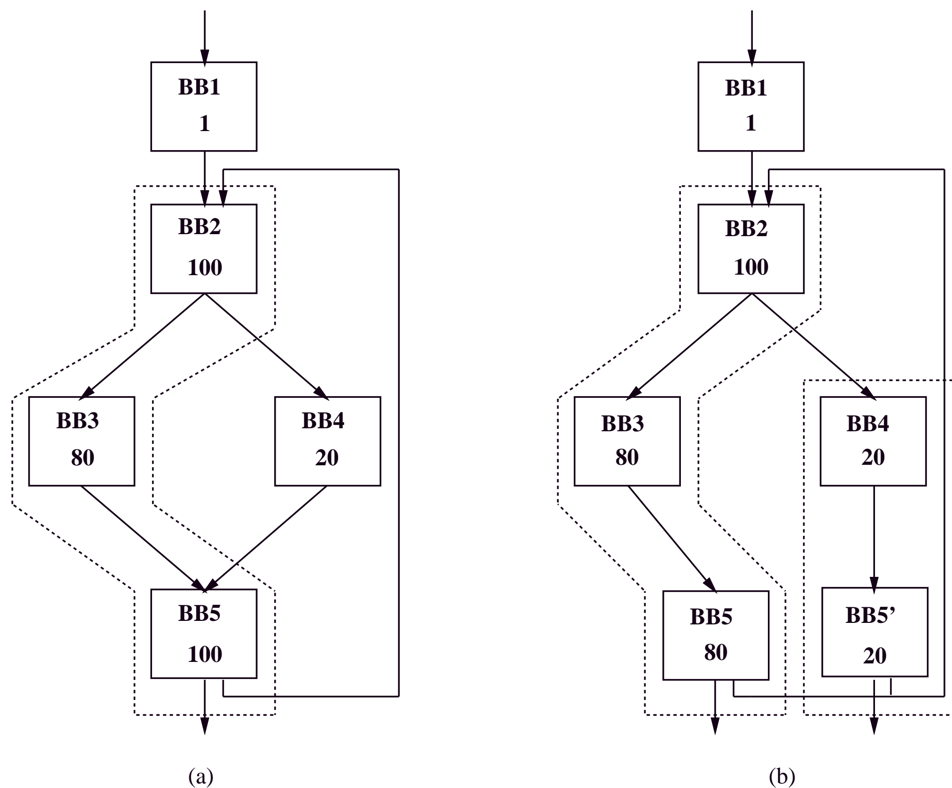

Figure 1: Example of superblock formation, (a) after trace selection, (b) after tail duplication.

A superblock is a sequence of consecutive instructions in which flow of control enters at the beginning but may leave at one or more exit points [13]. Superblocks are formed in two steps. First, sets of basic blocks which are likely to execute in sequence, traces [8] [7], are identified using execution profile information [3]. Then, tail duplication is performed to eliminate all side entrances into the trace. An example illustrating superblock formation is presented in Figure 1. The number associated with each basic block is its estimated execution frequency. In Figure 1a, the most likely execution path through the loop is selected as a trace, BB2, BB3, BB5. To convert the trace into a superblock, tail duplication is performed to eliminate the side entrance into the trace (see Figure 1b). After tail duplication, BB4 and BB5′ may also be combined to form another superblock.

2.2 Superblock Scheduling

Superblock scheduling is an extension of trace scheduling [8] which reduces some of the bookkeeping complexity [4]. 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 enforce two major restrictions on speculatively moving or percolating an instruction, J, before a branch, BR: (1) the destination of J is not used before it is redefined when BR is taken,[1] and (2) J 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.

2.3 Restricted Percolation Scheduling Model

The scheduler enforces both restrictions (1) and (2) when using the restricted percolation scheduling model [4]. Thus, only instructions which the compiler can guarantee to never cause exceptions are candidates for speculative code motion. For conventional processors, memory load, memory store, integer divide, and all floating point instructions are potentially excepting 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 potentially excepting

[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.

instructions with long latency, such as load instructions, above branches.

2.4 Instruction Boosting Scheduling Model

The scheduler enforces neither restriction when using the instruction boosting scheduling model [22] [21]. The restrictions are overcome by providing sufficient hardware storage to buffer results until the branches an instruction is moved above 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.

Exceptions for boosted instructions are handled by marking in the appropriate shadow structure whether an exception occurred during execution. At the excepting instruction's commit point, the contents of the shadow structure are examined to determine if an exception condition exists. If an exception condition exists, all information in the shadow structure is discarded and a branch is made to a compiler-generated recovery block. 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, an uncommitted speculative instruction may always be re-executed by retrieving its operands from the architectural register file. Finally, the exception is handled (either terminating program execution or recovering from the exception) using traditional exception handling techniques since the exception is regenerated in a sequential processor

state.

2.5 Ignoring Exceptions with the General Percolation Scheduling Model

The scheduler removes restriction (2) using the general percolation model [4]. Exceptions that may alter program execution are avoided by converting all speculative instructions which potentially cause exceptions into non-excepting 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 excepting opcodes. When an exception occurs for a silent instruction, the memory system or function unit simply ignores the exception and writes a garbage value into the destination register.[2] 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-excepting instruction which produces an exception [5]. The use of *NaN* is then signaled by any excepting 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 handling 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

[2]Note, exceptions such as page faults are handled immediately for silent instructions in the same manner as excepting instructions.

performance level of instruction boosting with unlimited boosting level [4] at 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 *potentially excepting instruction* (PEI). The sentinel reports any exceptions that were caused when the PEI is speculated. The sentinel can either be an existing instruction in the program or a newly created instruction. In the following subsections, the model of execution, the required architectural support, the algorithm for sentinel scheduling, and several other important issues are described.

3.1 Model of Execution

Conceptually, each instruction, J, 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 J can be speculatively executed, provided the sentinel part of J remains in J's home block. The *home block* of an instruction is the original basic block the instruction resides in before compile-time scheduling. The sentinel part of J can be eliminated if there is another

instruction, K, in J's home block which uses the result of J. The sentinel part of K will signal any exceptions caused by both J and K, which makes it a shared sentinel between J and K. Applying this argument one level further, if an instruction, L, in K's home block which uses the result of K can be found, its sentinel part may serve as the shared sentinel of J, K, and L. In this case, the semantics of K are defined so as to propagate an incoming exception from J to L's sentinel.

For each PEI, a recursive search may be applied to identify a tree of instructions which use its result. The search terminates along a path when an instruction that has no uses in its home block is encountered.[3] Such an instruction is termed an *unprotected instruction*. If all instructions in a PEI's tree of uses are speculatively executed, an explicit instruction must be created to act as the sentinel of the PEI. The explicit sentinel is restricted to remain in the PEI's home block.

Since some instructions may never result in exceptions, e.g., integer add, the sentinel part is not required for all instructions. An instruction only requires a sentinel part if it may cause an exception, or it is used to report an exception for a dependent PEI.

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 speculatively scheduled. A second extension is an exception tag added to each register in the register file. The

[3]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-execution of fewer instructions for recovery, and reducing register lifetimes.

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

† union of all source operand exception tags of J

‡ the first source operand of J whose exception tag is set

Table 1: Exception detection with sentinel scheduling.

exception tag is used to mark an exception that occurs when a speculative instruction is executed.[4] 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 during context switch.

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

Execution of a Speculative Instruction. When J is a speculative instruction, exceptions will not be signaled immediately. If all the source register exception tags of J are reset, conventional execution results when J does not cause an exception. When J does cause an exception, the exception tag of the destination register is set, and the program counter (PC) of J is copied into the data field of the destination register. The PC of J 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

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

latency function units [5] [19]. If one or more of the source register exception tags of J are set, an exception propagation occurs. This is independent of whether J 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 J 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 J is not a speculative instruction, conventional execution results if all source registers have their exception tags reset. When J causes an exception, the exception is signaled immediately, and J 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 J 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 $check(reg)$. This instruction is inserted as the explicit sentinel when no use of a speculative PEI exists in its home block. 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 [15].

3.3 Sentinel Superblock Scheduling Algorithm

As previously discussed, superblock code scheduling consists of two major steps, dependence graph construction and list scheduling. The dependence graph contains dependence arcs to represent all data and control dependences between instructions in the superblock. With the sentinel scheduling model, only restriction (1) (Section 2.2) is enforced for inserting control dependences. Therefore, a control dependence arc from a branch instruction, BR, to another instruction, J, is inserted if the location written to by J is used before being redefined when BR is taken. This is the same restriction applied using the general percolation scheduling model. As with general percolation, memory stores are not allowed to be speculative. However in Section 5, an extension to remove this constraint will be discussed.

Prior to list scheduling, an additional step is added for sentinel scheduling. In this step, potential sentinels are identified for each PEI the scheduler is allowed to speculate. In general, any instruction from a PEI's home block which uses the result of the PEI is a potential sentinel. However, a simplifying assumption to recognize potential sentinels only along one path in the dependence graph is made. This assumption is utilized to reduce the complexity associated with exception recovery (Section 4). The overhead associated with limiting the number of potential sentinels is that a larger number of explicit sentinels may be inserted than are required. This overhead is discussed further in Section 6.

An algorithm to identify potential sentinel instructions for all PEI's in a superblock is presented in Figure 2. For each PEI, a leaf node in the dependence subgraph is identified. Then all instructions along that path are marked as potential sentinels for the PEI. Instructions with a successor in the chain are marked as protected. Protected instructions may be freely speculated since the next

```
identify_potential_sentinels(superblock) {
    /* initialization, mark all instructions as protected */
    for each instruction in superblock, J {
        J→protected = 1
        J→sentinel = NULL
    }
    /* identify potential sentinels */
    for each instruction in superblock, J {
        if (J not allowed to be speculative)
            continue
        if ((J is unprotected) OR (J is potentially excepting)) {
            use = instruction in home_block(J) such that there is a flow
                dependence from J to use
            /* use in home block serves as the sentinel for J if J is speculated */
            if (use) {
                J→protected = 1
                use→protected = 0
                J→sentinel = use
                /* Do not allow potential sentinel to move to subsequent block */
                add control dependence from use to use→post_branch
            }
            /* No use in the home block so instruction is marked as unprotected */
            else {
                J→protected = 0
            }
        }
    }
    /* Identify last potential sentinel for each PEI which may be speculative */
    for each instruction in superblock, J {
        if (J is potentially excepting) {
            last = J
            while (last→sentinel)
                last = last→sentinel
            J→last_potential_sentinel = last
        }
    }
}
```

Figure 2: Algorithm to identify potential sentinel instructions.

instruction in the chain will check all exceptions propagated or caused by the instruction. The last instruction in the chain is marked as unprotected. If an unprotected instruction is speculated, an explicit sentinel must be created by the scheduler to check all exception conditions propagated through the chain. The last instruction in the chain is also recorded as the last potential sentinel for the PEI.

A modified form of list scheduling for superblocks to insert the necessary explicit sentinels is then performed as the final step of sentinel scheduling. The algorithm used is presented in Figure 3 with the additions to the basic superblock scheduling algorithm in bold type. The only modification required for sentinel scheduling is to insert an explicit sentinel instruction when an unprotected instruction is speculated. The explicit sentinel is restricted to be scheduled in the instruction's home block by adding the appropriate control dependences. Note that the algorithm contains two function calls for handling exception recovery. Exception recovery with sentinel scheduling is discussed in Section 4.

3.4 Sentinel Scheduling Example

To illustrate sentinel scheduling and exception detection with sentinel scheduling, consider the assembly code fragment shown in Figure 4(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. In the example, potentially excepting instructions B and C may be speculated, therefore a sentinel instruction must be kept in the home block of B and C to check their exception status if they are speculated.

The potential sentinels for B are identified as D and F. Since F is the last use in the chain of

```
schedule(superblock) {
    build_dependence_graph(superblock)
    identify_potential_sentinels(superblock)
    clear resource_usage_map
    issue_time = 0
    while (unscheduled set of instructions is not empty) {
        issue_time += 1
        active_set = set of unscheduled instructions that are ready
        sort active set according to instruction priority
        for each instruction in active_set, J {
            if (not all resources J requires are free)
                continue
            if ((enable_recovery) AND (! compatible_with_active_intervals(J)))
                continue
            /* J is scheduled at issue_time */
            mark required resources of J busy in resource_usage_map
            delete J from set of unscheduled instructions
            J→issue_time = issue_time
            if (J is speculative) {
                set speculative modifier of J
                /* create an explicit sentinel if speculate an unprotected instruction */
                if (J is unprotected) {
                    create a new instruction, check(dest(J))
                    J→sentinel = check
                    add flow dependence J to check
                    /* Restrict explicit sentinel to remain in J's home block */
                    add control dependence from J→prev_branch to check
                    add control dependence from check to J→post_branch
                    insert check into set of unscheduled instructions
                }
            }
            if (enable_recovery) update_intervals(J)
            /* check for control-flow hazards associated with an downward code motion */
            for each branch J moved below in superblock, BR {
                if (J→dest not live when BR is taken)
                    continue
                insert a copy of J into target superblock of BR
            }
        }
    }
}
```

Figure 3: Sentinel superblock scheduling algorithm.

flow dependences, it is marked as unprotected (Figure 4(a)). Similarly, the potential sentinel for C is E, which is unprotected since it is the last use in this chain. The code segment after scheduling is shown in Figure 4(b). Four instructions (B, C, D, and E) are moved above the branch (A), therefore their speculative modifiers are set. Instruction E, though, is unprotected, so an explicit sentinel (G) must be inserted into E's home block to check the exception condition of C. In the final schedule, instructions F and G serve as sentinels for the potentially excepting instructions B and C, respectively.

An execution sequence for the scheduled code segment in which instruction B causes an exception is shown in Figure 5. 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.

```
  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
```

(a)

```
∗ B[1]: r1 = mem(r2+0)
∗ C[1]: r3 = mem(r4+0)
∗ D[2]: r4 = r1+1
∗ E[2]: r5 = r3×9
  A[2]: if (r2==0) goto L1
‡ F[3]: mem(r2+0) = r4
‡ G[3]: check(r5)
∗      speculative instruction
‡      sentinel
[n]    indicates in which cycle the instruction is executed
```

(b)

Figure 4: Example of sentinel scheduling. (a) Original program segment, (b) Program segment after scheduling.

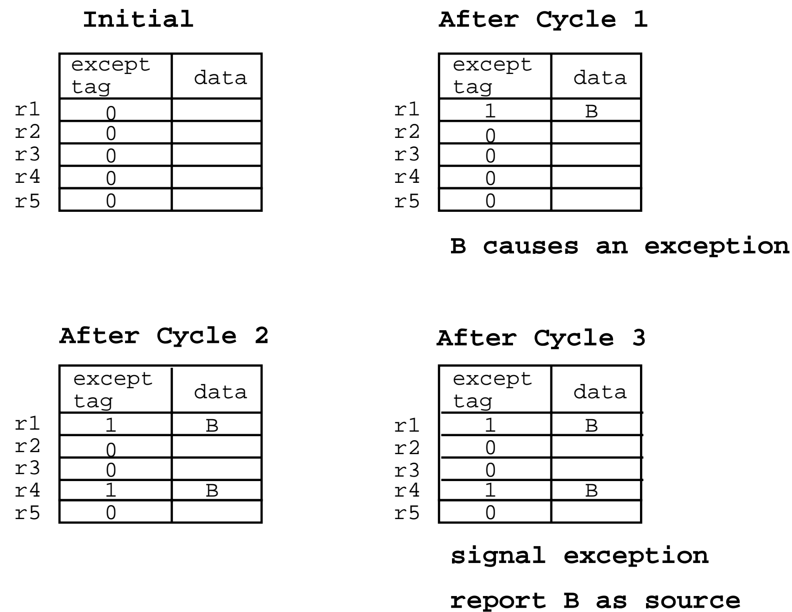

Figure 5: Example of exception detection using sentinel scheduling.

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 referenced before being defined in a function may have their exception tag set from the execution of a previous function or program. 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 [1], and inserts additional instructions into the beginning of a function to reset the exception tags of the corresponding registers. Therefore spurious errors associated with referencing uninitialized registers or variables are prevented.

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.

4 Exception Recovery

For many types of exceptions, it is desirable to recover from the exception rather than abort program execution. Recovery generally consists of repairing the excepting instruction and continuing program execution. Recovery with speculative instructions is difficult because the exception condition may not be raised until long after the instruction is executed. Also, other speculative instructions which use the result of the excepting instruction are likely to have executed. Therefore, when the exception is detected and repaired, a chain of dependent speculative instructions requires re-execution to generate a correct program state. The scheduling model must ensure that the excepting instruction and all dependent instructions are re-executable up to the point where the exception condition is checked.

4.1 Recovery Model

The compiler support required to ensure recovery is dependent on the recovery model utilized. In this paper, the recovery model assumed is as follows. A sentinel which detects an exception condition sets the processor PC to the excepting instruction's PC. The processor then enters an exception handling state which terminates when the execution reaches the sentinel again. The excepting instruction is re-executed as a non-speculative instruction to regenerate the exception

condition. After the exception is repaired, re-execution of the subsequent instructions proceeds.

Not all instructions between the excepting speculative instruction and the sentinel require re-execution. Instructions which are not re-executed are simply discarded. The minimal set of instructions which must be re-executed are those which are flow dependent on the excepting instruction. Flow dependent instructions propagate the exception condition for the excepting speculative instruction and therefore must be re-executed to obtain the correct result. Any superset of the flow dependent instructions may be chosen for re-execution. However, the re-execute set must be known by the compiler to ensure proper recovery.

Those non-speculative instructions thus re-executed are done so as normal non-speculative instructions (e.g., if they produce an exception, the exception is signaled immediately). Those speculative instructions thus re-executed are done so as normal speculative instructions (e.g., if they produce an exception, no exception is signaled but the exception tag and data field of the excepting instruction's destination register are set appropriately) with one modification. Transparent exceptions must be handled immediately for speculative instructions in the exception handling state. Transparent exceptions are exceptions such as page faults and TLB misses which occur independent of the program logic. This is necessary to ensure that speculative instructions which did not except in their original execution, produce the correct result during re-execution. When execution reaches the original sentinel instruction again, the exception handling state is exited and normal execution resumes.

Note that other models of recovery may be utilized in conjunction with sentinel scheduling. One effective alternative is recovery blocks [21]. In this model, the compiler generates the exact sequence of instructions that must be re-executed when an exception is detected by a particular sentinel. The advantage of this scheme is reduced complexity in the exception handling state. The

disadvantage of this scheme is static code size increases due to the recovery blocks.

4.2 Restartable Instruction Interval

In order to ensure recovery with the recovery model discussed in the previous section, each PEI which is speculated and its sentinel must delineate the endpoints of a restartable instruction interval. The interval consists of two types of instructions based on their execution status during recovery, those which are re-executed (RE) and those which which are not re-executed (NRE). An instruction interval is restartable if all elements of the interval satisfy the following constraints. First, none of the instructions in the interval may prevent re-execution of the RE instructions in the interval. These instructions will be referred to as *irreversible instructions*. For the purposes of this paper, an irreversible instruction has one of the three following properties: the instruction destroys the exception tag of a live register, the instruction modifies an element of the processor state which causes intolerable side effects, or the instruction cannot be executed more than one time. As a result, synchronization, I/O, and subroutine call instructions break restartable intervals.[5] Based on these properties, memory stores are not considered irreversible instructions.

The second constraint is that the operands of all RE instructions which are live at the start of the interval are not overwritten by any instruction in the interval. An operand is live if it is used by an RE instruction before it is defined by an RE instruction in the interval. The live set of an interval is calculated using techniques for standard dataflow analysis [1]. However, only the RE instructions in the interval are considered in the process. An algorithm to compute the live information for an instruction interval is presented in Figure 6. Live information is computed for both register and memory operands since both must be maintained to ensure a restartable interval.

[5] Note that if an architecture provides a means for the compiler to save/restore exception tags, a subroutine call alone is not an irreversible instruction. However, in this discussion it is assumed subroutine calls are irreversible.

```
update_interval_live_set(interval, J) {
    if (J is in interval→RE) {
        if (src_i(J) not in interval→def)
            interval→use = interval→use + {src_i(J)}
        if ((J is a memory load) AND (J not in interval→mem_def))
            interval→mem_use = interval→mem_use + {J}
        interval→def = interval→def + {dest_i(J)}
        if (J is a memory store)
            interval→mem_def = interval→mem_def + {(J)}
    }
}
```

Figure 6: Algorithm to calculate live information for an instruction interval.

PEI,RE	A: r1 = mem(r2)	PEI,RE	A: r1 = mem(r2)
NRE	B: r3 = r4+1	NRE	B: r3 = r4+1
RE	C: r4 = r1×r5	NRE	C: r4 = r1×r5
RE	D: r6 = r3+r4	RE	D: r6 = r3+r4
RE	E: r4 = mem(r3)	RE	E: r4 = mem(r3)
RE	F: r7 = r1-1	RE	F: r3 = r1-1
Sentinel	G: check(r4)	Sentinel	G: check(r4)
	(a)		(b)

Figure 7: Example of (a) A restartable instruction interval, (b) A non-restartable instruction interval.

To illustrate the computation of an interval live set, consider the example in Figure 7(a). The register live set consists of $r2$, $r3$, and $r5$. Although $r3$ is computed by B before it is used in the original execution of the interval. Instruction B is an NRE instruction, therefore $r3$ will not be re-defined during re-execution. Consequently, its contents must be preserved all the way to the end of the interval (during the original execution) and from the beginning of the interval (during re-execution) to ensure that D may re-execute correctly. Also, note that even though B uses $r4$ before it is defined, $r4$ is not included in the live set. This is because B is an NRE instruction. Therefore, its inputs operands may be modified without affecting the restartability of the interval.

The instruction interval shown in Figure 7(a) is thus restartable since none registers in the live set are modified in the interval. An example interval which is not restartable is shown in Figure 7(b). The register live set consists of $r2$, $r3$, and $r4$. The conditions for restartability are violated by two instructions in the interval. Instruction E overwrites $r4$ which prevents D from properly re-executing. Similarly, F overwrites $r3$ which prevents D and E from properly re-executing.

The compiler must maintain a restartable instruction interval for all PEI/sentinel pairs that are generated to ensure exception recovery may be performed. From the point of view of individual instructions, the compiler must satisfy the restartability constraints for all intervals which span an instruction. In the remainder of this section the required scheduler and register allocator support to maintain restartable instruction intervals is presented. Also, a discussion of various recovery models and the recovery model utilized for the experimental evaluation is presented.

4.3 Scheduler Support

Several additional restrictions must be added to the instruction scheduler to ensure all instruction intervals are restartable. The additional restrictions are as follows.

1. A speculative instruction cannot be moved beyond any irreversible instruction.
2. Exceptions for speculative instructions are not propagated across irreversible instructions.
3. A speculative instruction may not modify any of its own input operands.
4. All instructions which overwrite an operand in an instruction interval's live set may not be scheduled in the interval.

The first two scheduling restrictions are used to prevent an irreversible instruction from being included in any instruction interval. The first restriction is handled by inserting additional control dependences during dependence graph construction. A dependence arc is inserted from each irreversible instruction to all subsequent instructions in the superblock. The second restriction is maintained by modifying the definition of home block to account for irreversible instructions. Each irreversible instruction defines an additional basic block boundary as far as the scheduler is concerned. In this manner, the identify_potential_sentinels algorithm (Figure 2) will not search beyond an irreversible instruction for flow dependent instructions.

The last two scheduling restrictions are used to ensure the live operands of all RE instructions in the interval are not destroyed by any instruction in the interval. Compile-time renaming is utilized to overcome the third restriction. The destination of an instruction which may be speculated and overwrites one of its source operands (self anti dependence) is renamed to a new register. All uses

of the original register are then replaced with the renamed register. If necessary, a copy instruction is inserted to restore the proper value of the original register.

The fourth restriction is overcome by modifying the sentinel superblock scheduling algorithm. The modifications employ two functions that check and update the status of instruction intervals. The calls to these functions are included in the sentinel superblock scheduling algorithm shown in Figure 3. The first function is used to determine if scheduling an instruction at the current time is compatible with all active intervals. An instruction interval is active at the current time if the start of the interval has been scheduled and the end of the interval has not been scheduled. An instruction is compatible with an active interval if the interval remains restartable when the instruction is added. An NRE instruction is compatible by default since there are no restrictions on the redefinition of its input operands.

An RE instruction is compatible if all instructions which modify any of its input operands that are live in the interval may be scheduled after the end instruction of the interval. Instructions which modify live operands are identified by traversing the anti, output, memory anti and memory output dependences of a candidate RE instruction. If the modifying instruction is independent of the interval ending instruction it can be scheduled after the interval end point without a problem. However, if the modifying instruction is dependent on the instruction which ends the interval, the restartability of the interval may be maintained only by breaking the interval. An interval is broken by selecting an earlier potential sentinel for the end point. The new interval end point must be independent of the modifying instruction. Therefore, the modifying instruction must be able to be scheduled in the home block. If dependences prevent scheduling the modifying instruction in the home block, the candidate instruction is not allowed to be scheduled at the current time. An algorithm to determine if an instruction is compatible with all active intervals is presented in

```
compatible_with_active_intervals(J) {
    /* Create a temporary interval for J if it is a speculated PEI */
    if ((J is speculative) AND (J is potentially excepting)) {
        create a new active interval, temp
        temp→start = J
        temp→end = last potential sentinel of J
        temp→RE = temp→NRE = {}
    }
    compatible = 1
    for each active interval, interval {
        if (J in NRE for interval) continue
        /* Save the contents of all fields of interval, so they can be later restored */
        original = copy all elements of interval
        interval→RE = interval→RE + {J}
        update_interval_live_set(interval, J)
        /* Determine if any of J's operands in the use set of the interval are modified by
                instructions which cannot be scheduled outside the interval */
        for each dependence arc out of J, dep {
            if (((dep→type is anti or output) AND (dep→operand in interval→use)) OR
                    ((dep→type is memory anti or memory output) AND (dep→to_instr in interval→mem_use))) {
                /* An instruction not dependent on the end of the interval may always be moved
                        after the end of the interval to satisfy the dependence constraint */
                if (there is no dependence path from dep→to_instr to interval→end)
                    continue;
                /* Otherwise, the interval can be broken if dep→to_instr can be moved into
                        the home block of instruction which ends the interval */
                else if (there is a dependence path from dep→to_instr to interval→prev_br) {
                    compatible = 0
                    break
                }
            }
        }
        restore contents of interval with original
    if (! compatible) break
    }
    delete temp
    return (compatible)
}
```

Figure 8: Algorithm to determine if an instruction is compatible with all active intervals.

Figure 8.

The last modification to the sentinel superblock scheduling algorithm is a function which updates the contents of all active intervals after each instruction is scheduled. The algorithm used to update the active intervals is shown in Figure 9. When a PEI is scheduled speculatively, a new active interval is created. The interval begins with the PEI and its end point is set to the last potential sentinel of the PEI. The last potential sentinel is selected as the end point of the interval to ensure the interval is restartable for whichever potential sentinel is selected as the actual sentinel of the PEI. Since the last sentinel is the instruction which ends the chain of flow dependent potential sentinels, enforcing all scheduling restrictions to the last potential sentinel is sufficient to guarantee restartability.

The update algorithm also prevents instructions from overwriting the operands live in the interval by inserting additional dependence arcs. Similar to the previous algorithm, instructions which modify live source operands are identified by traversing the anti, output, memory anti, and memory output dependences for a new instruction added to a interval. A dependence arc is added from the interval end point to the modifying instruction to restrict the modifying instruction from entering the interval. If the modifying instruction is dependent on the end of the interval, the interval must be broken up. This is necessary to prevent a circular dependence condition between the modifying instruction and the end of the interval. The potential sentinel farthest down in the chain of flow dependences that is not dependent on the modifying instruction is selected as the new end point. If no such instruction exists, an explicit sentinel is created to serve as the end point of the interval.

An example to illustrate the handling of the scheduling restrictions is presented in Figure 10. For this example assume each instruction requires 1 cycle to execute, the processor has unlimited

```
update_intervals(J) {
    /* Create a new interval for a speculated PEI */
    if ((J is speculative) AND (J is potentially excepting)) {
        create a new active interval, interval
        interval→start = J
        interval→end = last potential sentinel of J
        interval→RE = interval→NRE = {}
        interval→use = interval→def = interval→mem_use = interval→mem_def = NULL
    }
    /* De-activate all intervals which end with J, note that J must be
            non-speculative to end an interval */
    for each active interval, interval {
        if (interval→end==J) deactivate interval
    }
    /* update all active intervals with J */
    for each active interval, interval {
        /* The recovery model utilized defines if J is an RE or NRE instruction for each interval */
        if (J in NRE for interval) {
            interval→NRE = interval→NRE + {J}
            continue
        }
        interval→RE = interval→RE + {J}
        update_interval_live_set(interval, J)
        for each dependence arc out of J, dep {
            if (((dep→type is anti or output) AND (dep→operand in interval→use)) OR
                    ((dep→type is memory anti or memory output) AND (dep→to_instr in interval→mem_use))) {
                if (there is a path in the dependence graph from dep→to_instr to interval→end) {
                    /* break up the interval to satisfy the dependence constraint */
                    S = farthest instruction in the flow dependence chain of potential sentinels for
                            interval→start that is not dependent on dep→to_instr
                    if (S is not speculated) {
                        mark S as unprotected
                        interval→end = S
                    }
                    else {
                        /* Create an explicit sentinel since all potential sentinels for the broken
                                interval have been speculated */
                        create a new instruction, check(dest(S))
                        add a flow dependence from S to check
                        add a control dependence from S→prev_branch to check
                        add a control dependence from check to S→post_branch
                        interval→end = check
                        add check into set of unscheduled instructions
                    }
                }
                insert a dependence of type dep→type between interval→end and dep→to_instr
            }
        }
    }
}
```

Figure 9: Algorithm to update all active intervals.

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

   †   instruction considered for speculative execution
   ‡   last potential sentinel for D
   *   speculative instruction
   ◇   sentinel for D
   [n] indicates in which cycle the instruction is executed

         (a)                           (b)
```

Figure 10: Example of sentinel scheduling to ensure recovery. (a) Original program segment. (b) Program segment after scheduling.

resources, and only memory load and store instructions may cause exceptions. It is further assumed that all instructions in an interval are RE instructions. The first restriction is for A. Instruction A is an irreversible instruction, therefore no speculative code motion is allowed across it. Instruction D may be speculated provided several constraints are observed. First, H overwrites a source operand of D. Therefore, H must be scheduled after the end point of the interval started by D, namely G. Similarly, if the compiler cannot determine that instructions D and F access different memory locations, F must be scheduled after G (due to memory anti-dependence).

Instruction E may also be speculated in the example. Instruction E is self anti-dependent, therefore the destination of E must be renamed to a new register ($r10$ in the example). All uses of the original register $r2$ are also renamed to $r10$ and a copy instruction, I, is inserted assuming $r2$ is live outside the code segment. Let E' be the instruction derived from E by renaming $r2$ to $r10$. Since the copy instruction I is anti-dependent on E', the copy is restricted to be scheduled after the end point of all intervals which contain E'. Thus, I is scheduled after G. The final schedule

with all restrictions observed is shown in Figure 10(b). Notice that if speculative instruction D causes an exception, the exception is detected by its sentinel G. Instructions D, B, E', and C are then re-executed during exception handing mode. All instructions re-execute correctly since their source operands have not been destroyed.

Additional compile-time renaming is also effective to minimize the number of anti-dependences that must be enforced during scheduling due to the fourth restriction. In our current implementation, anti and output dependence removing transformations are applied to superblocks prior to scheduling [13].

4.4 Register Allocator Support

The register allocator must also be modified to ensure that all PEI/sentinel intervals are restartable. The following additional restrictions must be utilized by the register allocator to ensure exception recovery is possible.

1. The contents of a register in the live set of an interval may not be overwritten in the interval.
2. A destination register of an RE instruction may not be spilled in an interval.

The first restriction is handled by adding all instructions in an interval to the liverange of each register in the interval's live set. By extending the liverange of a register across all intervals in which it is live, the register contents are preserved across the necessary instructions to ensure restartability of all intervals. The algorithm to construct liveranges of each virtual register is augmented to add the contents of the intervals which the register is live. Traditional graph coloring may then be applied to achieve the desired allocation.

In the example shown in Figure 10(b), assuming all instructions are re-executed during exception recovery, the live set of interval from D to G consists of $r2$, $r3$, and $r6$. All instructions in the interval are thus added to the liveranges of these registers. As a result, even though D may be the last use of $r6$, the register allocator may not re-use the physical register mapped to $r6$ until after G.

The first restriction also implies that register source operands of speculative PEIs may not be spilled to memory by the register allocator. This is necessary with the recovery model used in this paper because during exception handling the processor does not know to re-execute spill load instructions to restore appropriate spilled source register operand values. In the current implementation, the register allocator enforces this restriction by de-speculating a speculative instruction whose source operands are spilled. De-speculation or downward code movement back to the PEI's home block is performed incrementally until either the live range becomes allocatable or the instruction's home block is reached. At the point when the home block is reached, the instruction is no longer speculative, and the register allocator is free to spill its source operands.

The second restriction is necessary to ensure improper exceptions are not signaled when a speculative instruction's destination is spilled to memory. In order to spill the register, a store instruction will read the contents of the speculative instruction's destination register. If that register contains an exception condition, an exception will be signalled. However, execution may not reach the home block of the speculative instruction. Therefore, an improper exception signal may occur. Preserving destination registers of speculative instructions may be achieved using the same de-speculation process. Speculative instructions whose destination live range cannot be allocated are incrementally moved downward until either the live range becomes allocatable or the speculative instruction's home block is reached. Again, once the home block is reached, the instruction is no

longer speculative and the register allocator is free to spill its destination operand.

Note that if the architecture provides a special set of spill instructions, the second restriction for register allocation may be eliminated. The spill instructions must save/restore the exception tag along with the data contents of the register. Furthermore, the spill instruction which saves the contents of a register must ignore the exception status of the register to prevent signalling improper exceptions. Finally, the spill instructions must be included in the RE sets of all intervals which span them to ensure updated values are placed on the stack during re-execution.

The current implementation of the scheduler does not utilize any information regarding register usage to guide the schedule. Therefore, in superblocks with a large amount of register pressure, the register allocator will be required to de-speculate many speculative instructions to satisfy the restrictions for exception recovery. More advanced scheduling techniques which integrate parts of register allocation and scheduling may be used to achieve a more efficient schedule [10] [9]. Currently, these techniques are being studied to improve the performance of sentinel scheduling in regions with large register pressure.

To summarize, by enforcing constraints for scheduling and register allocation, one can guarantee that all speculative PEI/sentinel pairs form a restartable instruction interval. Therefore, an exception for a speculative instruction may be repaired and all RE instructions in the interval started by the PEI may be re-executed to achieve a correct program state. The overhead associated with enforcing these constraints is reduced scheduling freedom caused by additional dependence constraints and speculation limits imposed by the register allocator. Also, additional instructions to accomplish renaming are typically necessary. The overhead of ensuring exception recovery with sentinel scheduling will be evaluated in Section 6.

5 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.

5.1 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.[6] 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

[6] 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.

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.

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.

5.2 Scheduling Support for Store Movement

An instruction scheduler can be extended to move store instructions above branch instructions in a straight-forward manner. Stores are permitted to move above branches by removing control dependences between a store instruction and all preceding branch instructions in a superblock during dependence graph construction. All store instructions are marked unprotected by the identify_potential_sentinels algorithm (Figure 2) since store instructions have no destination register. Finally, list scheduling is modified to insert confirm_stores rather than checks as explicit sentinels for stores. Also, the scheduler must set the index field of the confirm_store when a store is speculated.

The value of the 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 have all 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.

Exception recovery is also possible with speculative stores. The only modification required is to allow re-executed speculative stores to replace their corresponding probationary entry in the store buffer. This is necessary for two reasons. First, multiple store buffer entries are not allowed for a speculative store which is re-executed several times. Second, the order stores are inserted into the buffer must not be altered from the order the compiler calculated during scheduling to ensure proper confirmation.

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.

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 3: Instruction latencies.

6 Experimental Evaluation

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

6.1 Methodology

Sentinel superblock scheduling has been incorporated 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 [4]. 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 in-order execution with register interlocking similar to the CRAY-1 [20]. The instruction set is a superset of the the HP PA-RISC instruction set with extensions to support sentinel scheduling [11]. Instruction latencies of the HP PA-RISC 7100 (see Table 3) are assumed. The basic processor has 64 integer registers, 64 single precision floating point registers which can accommodate 32 double precision values, and an 8 entry store buffer. The basic processor is as-

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

Table 4: Benchmarks.

sumed 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 are the 14 non-numeric programs shown in Table 4. The benchmarks consist of 5 programs from the SPECint92 suite and 9 other commonly used non-numeric programs.

6.2 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. With sentinel scheduling, a variation of the recovery model discussed in Section 4.1 is utilized. In this variation, the RE set of each instruction interval consists of all speculative instructions in the interval. Therefore, during recovery only

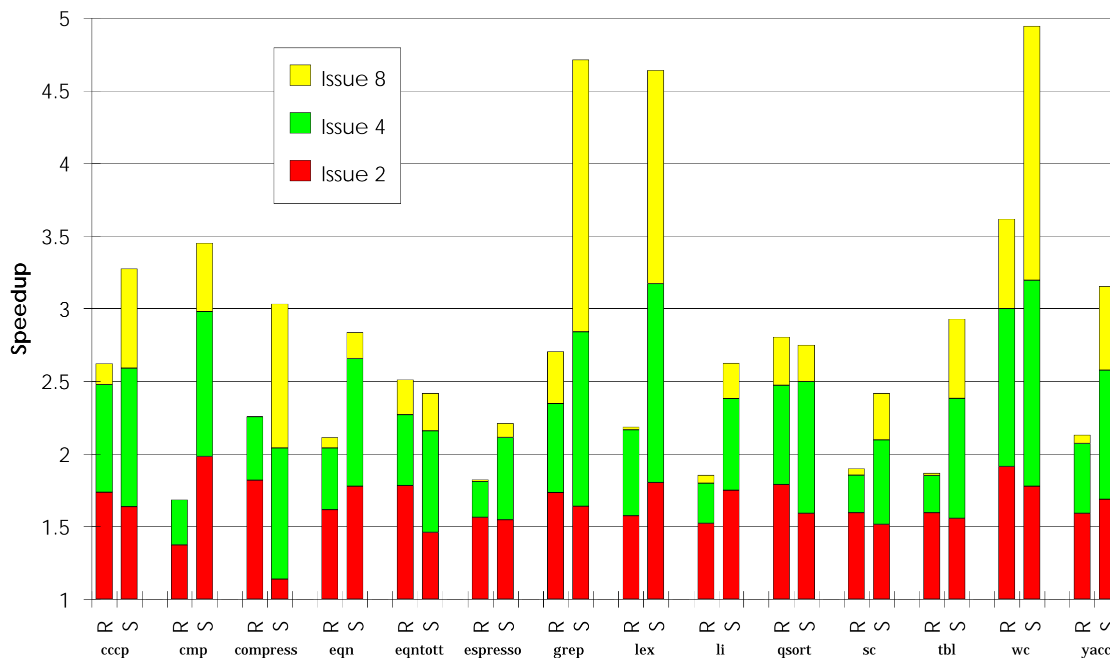

Figure 11: Performance comparison of sentinel scheduling (S) and restricted percolation (R) with 64 integer and 64 floating-point registers.

speculative instructions are re-executed. The performance evaluation of alternative recovery models is beyond the scope of this paper.

Comparison of Sentinel Scheduling and Restricted Percolation. The performance of the sentinel scheduling model and the restricted percolation scheduling model is compared in Figure 11. The base configuration for speedup calculations in this graph is an issue-1 processor with restricted percolation code scheduling. Note that the results for sentinel scheduling include the scheduling and register allocation constraints to ensure exception recovery is possible.

In general, sentinel scheduling provides large performance improvements over restricted percolation for 4-issue and 8-issue processors. The largest speedups are achieved for *cmp*, *grep*, and *lex*. The ability to speculatively execute PEIs 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 is extremely important for VLIW and superscalar processors. The importance of speculating PEIs also grows for higher issue rate processors. Higher issue rate processors require larger amounts of ILP to fully utilize the available resources. Therefore, the additional freedom to speculate PEIs enables the compiler to more effectively utilize the available processor resources.

Performance loss with sentinel scheduling on all processor configurations is observed for two benchmarks, *eqntott* and *qsort*. The major reason for the performance loss is the overhead associated with extra instructions inserted to remove anti-dependences. For example, in the current implementation, all self anti-dependent instructions are split into 2 instructions to enable speculative execution. Another reason is the additional constraints placed on speculative code motion with sentinel scheduling. In order to ensure exception recovery is possible, restrictions are placed on both potentially excepting instructions and non-excepting instructions. These additional restrictions limit the scheduling freedom of non-excepting instructions with sentinel scheduling.

Evaluation of the Overhead for Exception Detection. To evaluate the overhead of exception detection alone for sentinel scheduling, the performance of sentinel scheduling without recovery constraints is compared with the general percolation scheduling model. Since general percolation provides no support for exception handling, general percolation provides an upper limit on the performance of sentinel scheduling. For all benchmarks and issue rates, the sentinel scheduling incurs a maximal performance overhead of 2% with 64 integer and 64 floating-point register with 64 integer and 64 floating-point registers. The same overhead was also observed for 32 and 48 register configurations. The small overhead indicates that few explicit sentinels must be inserted to allow PEIs to execute speculatively. This is confirmed with Table 5 which shows

Benchmark	Assembly Instructions	Check Instructions
cccp	13635	19
cmp	778	25
compress	3709	19
eqn	11174	5
eqntott	11363	87
espresso	60153	242
grep	4470	2
lex	18089	70
li	19061	93
qsort	998	7
sc	29348	177
tbl	22696	140
wc	599	0
yacc	26758	104

Table 5: Maximum number of explicit sentinels required with 64 integer and 64 floating-point registers.

the static number of checks inserted compared to the total number of static instructions in the program.

The results indicate that a non-speculative potential sentinel can almost always be found for a speculated PEI. Therefore, the use of a simplified algorithm to identify potential sentinels (Section 3.3) does not restrict performance significantly.

Evaluation of the Overhead of Exception Recovery. To evaluate the overhead of exception recovery, the performance of sentinel scheduling with recovery constraints and the general percolation scheduling model is compared with 64 integer and 64 floating-point registers in Figure 12. The same performance comparison is shown in Figures 13 and 14 with 48 integer and 48 floating-point registers and 32 integer and 32 floating-point registers, respectively. The graphs report the performance ratio achieved by a particular processor configuration for sentinel scheduling

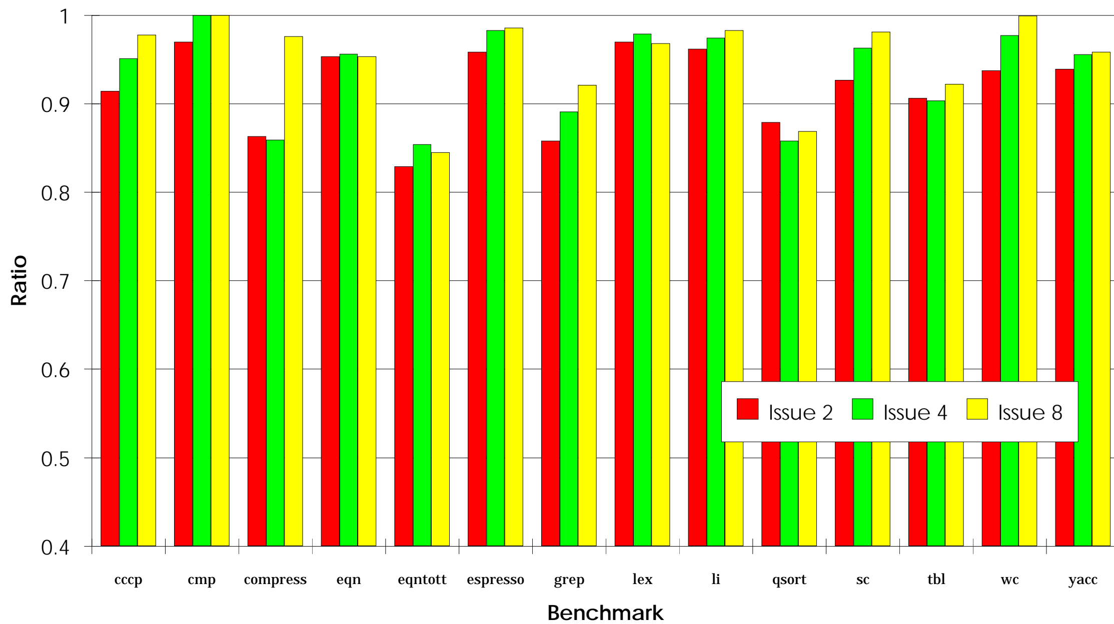

Figure 12: Performance comparison of sentinel scheduling and general percolation with 64 integer and 64 floating-point registers.

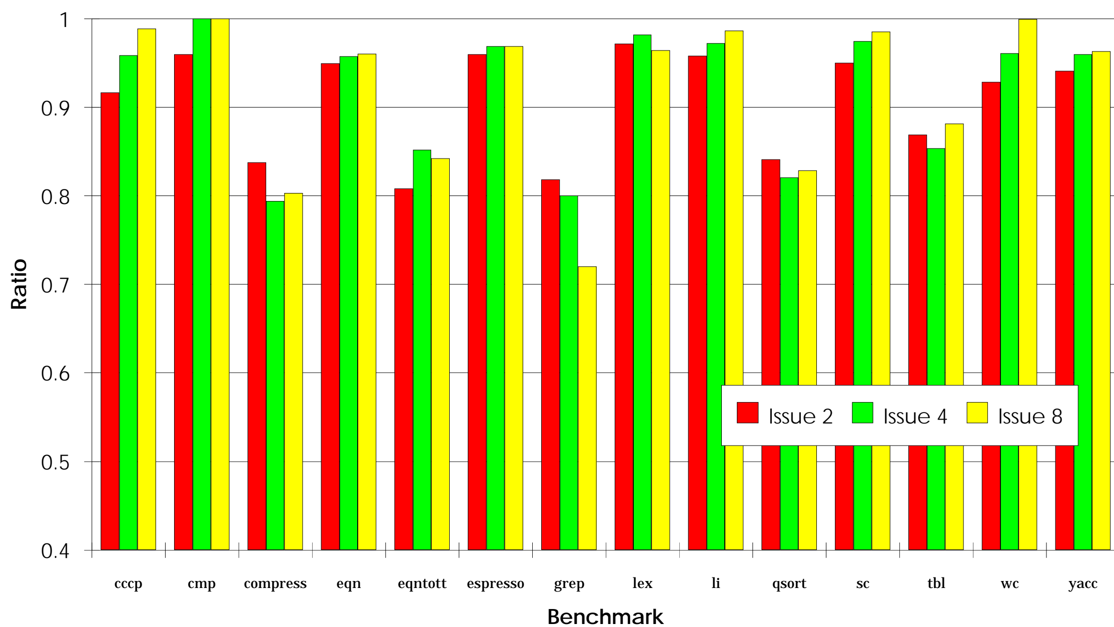

Figure 13: Performance comparison of sentinel scheduling and general percolation with 48 integer and 48 floating-point registers.

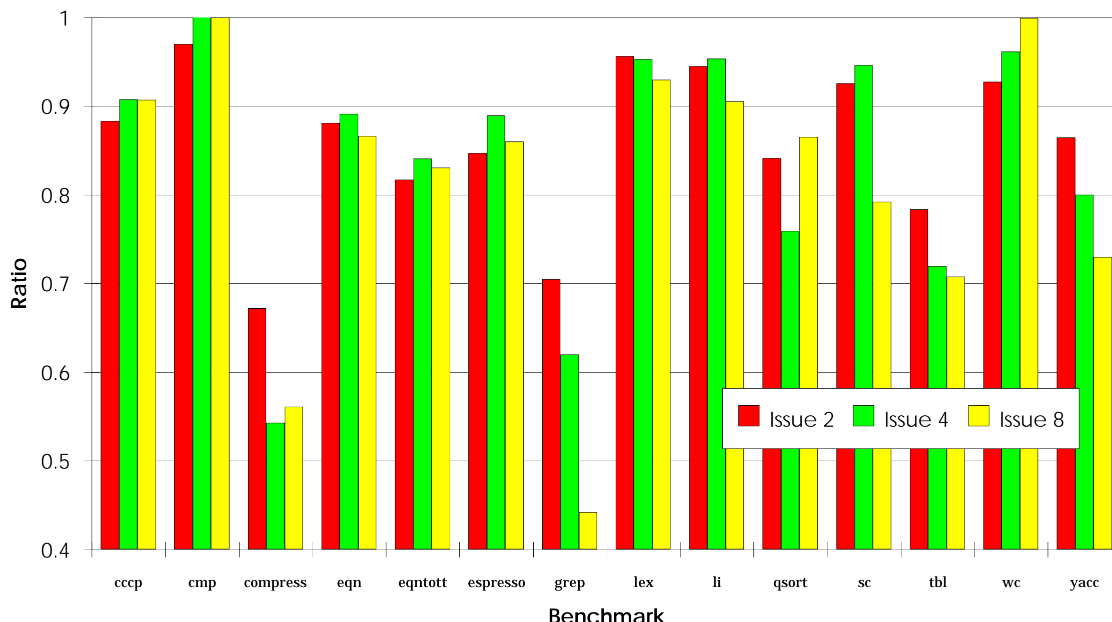

Figure 14: Performance comparison of sentinel scheduling and general percolation with 32 integer and 32 floating-point registers.

compared to general percolation. Again, since general percolation provides no support for exception handling, general percolation provides an upper limit on the performance of sentinel scheduling.

For the 64 register configuration (Figure 12), the performance of sentinel scheduling is extremely close to general percolation. The performance of sentinel scheduling is greater than 90% for all issue rates for ten of the fourteen benchmarks studied. This indicates that added register pressure incurred by sentinel scheduling is tolerated with 64 integer and 64 floating point registers. The remaining performance difference between sentinel scheduling and general percolation is due to additional instructions inserted for sentinel scheduling to ensure proper exception handling. These include explicit sentinels (checks) and instructions to remove self anti-dependences.

For the 32 register configuration (Figure 14), the performance overhead of sentinel scheduling is much larger. The largest performance losses occur for *compress* and *grep*. In both of these

benchmarks, the excessive register pressure forces the register allocator to de-speculate many instructions. Therefore, a much more serial schedule results which leads to performance loss. For the other benchmarks, however, only a moderate performance loss is observed when the register file is reduced from 64 to 32 registers. For, seven of the remaining twelve benchmarks, sentinel scheduling achieves more than 85% of the performance of general percolation.

For the 48 register configuration (Figure 13), the performance more closely follows that of the 64 register configuration. Therefore, the additional register pressure incurred by sentinel scheduling is tolerated for most benchmarks with 48 integer and 48 floating-point registers. However, the performance loss for *compress* and *grep* remains relatively large.

Effectiveness of Allowing Speculative Stores. The performance of sentinel scheduling with support for speculative stores is evaluated in [17]. In general, small to moderate performance gains were observed. An average of 7.4% improvement was observed with speculative stores for non-numeric benchmarks. The major reason for this improvement is that there are store instructions which limit the speculation of subsequent load instructions due to unresolvable memory dependences. However, by speculating the store instructions, the subsequent loads may also be further speculated, thereby reducing the dependence length.

7 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 excepting 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. Exception recov-

ery may also be utilized with sentinel scheduling by providing additional scheduling and register allocation support. The extra support ensures that all speculative instructions between an speculated potential excepting instruction form a restartable instruction interval and therefore may be re-executed.

Sentinel scheduling is shown to provide substantial performance improvements over restricted percolation for a set of non-numeric programs. Also for processors with latencies similar to those used in this paper, the performance of sentinel scheduling is shown to closely match the performance of general percolation with 64 integer and 64 floating-point registers. General percolation provides an upper limit on the performance of sentinel scheduling since no constraints associated with exception handling are utilized. The overhead associated with recovery becomes more evident with 32 integer and 32 floating-point registers. However, in most cases, only a moderate overhead is observed.

Acknowledgements

The authors would like to thank John Gyllenhaal, Thomas Conte, and Sadun Anik, 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.

References

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

[2] AIKEN, A., AND NICOLAU, A. Optimal loop parallelization. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (June 1988), pp. 308–317.

[3] CHANG, P. P., AND HWU, W. W. Trace selection for compiling large C application programs to microcode. In *Proceedings of the 21st International Workshop on Microprogramming and Microarchitecture* (November 1988), pp. 188–198.

[4] CHANG, P. P., MAHLKE, S. A., CHEN, W. Y., WARTER, N. J., AND HWU, W. W. IMPACT: An architectural framework for multiple-instruction-issue processors. In *Proceedings of the 18th International Symposium on Computer Architecture* (May 1991), pp. 266–275.

[5] COLWELL, R. P., NIX, R. P., O'DONNELL, J. J., PAPWORTH, D. B., AND RODMAN, P. K. A VLIW architecture for a trace scheduling compiler. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems* (April 1987), pp. 180–192.

[6] EBCIOGLU, K. A compilation technique for software pipelining of loops with conditional jumps. In *Proceedings of the 20th Annual Workshop on Microprogramming and Microarchitecture* (December 1987), pp. 69–79.

[7] ELLIS, J. *Bulldog: A Compiler for VLIW Architectures.* The MIT Press, Cambridge, MA, 1985.

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

[9] FREUDENBERGER, S., AND RUTTENBERG, J. Phase ordering of register allocation and instruction scheduling. In *Code Generation - Concepts, Tolls, Techniques* (May 1991).

[10] GOODMAN, J. R., AND HSU, W. C. Code scheduling and register allocation in large basic blocks. In *Proceedings of the 1988 International Conference on Supercomputing* (July 1988), pp. 442–452.

[11] HEWLETT-PACKARD CO. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual.* Hewlett-Packard Co., Cupertino, CA, 1990.

[12] HSU, P. Y. T., AND DAVIDSON, E. S. Highly concurrent scalar processing. In *Proceedings of the 13th International Symposium on Computer Architecture* (June 1986), pp. 386–395.

[13] HWU, W. W., MAHLKE, S. A., CHEN, W. Y., CHANG, P. P., WARTER, N. J., BRINGMANN, R. A., OUELLETTE, R. G., HANK, R. E., KIYOHARA, T., HAAB, G. E., HOLM, J. G., AND LAVERY, D. M. The Superblock: An effective structure for VLIW and superscalar compilation. *Journal of Supercomputing* (February 1993).

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

[15] KANE, G. *MIPS R2000 RISC Architecture.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1987.

[16] LAM, M. S. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (June 1988), pp. 318–328.

[17] MAHLKE, S. A., CHEN, W. Y., HWU, W. W., RAU, B. R., AND SCHLANSKER, M. S. Sentinel scheduling for superscalar and VLIW processors. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1992), pp. 238–247.

[18] RAU, B. R., AND GLAESER, C. D. 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* (October 1981), pp. 183–198.

[19] RAU, B. R., YEN, D. W. L., YEN, W., AND TOWLE, R. A. The Cydra 5 departmental supercomputer. *IEEE Computer* (January 1989), 12–35.

[20] RUSSELL, R. M. The Cray-1 computer system. *Communications of the ACM 21*, 1 (January 1978), 63–72.

[21] SMITH, M. D., HOROWITZ, M. A., AND LAM, M. S. Efficient superscalar performance through boosting. In *Proceedings of the Fifth International Conference on Architecture Support for Programming Languages and Operating Systems* (October 1992), pp. 248–259.

[22] SMITH, M. D., LAM, M. S., AND HOROWITZ, M. A. Boosting beyond static scheduling in a superscalar processor. In *Proceedings of the 17th International Symposium on Computer Architecture* (May 1990), pp. 344–354.

[23] TIRUMALAI, P., LEE, M., AND SCHLANSKER, M. Parallelization of loops with exits on pipelined architectures. In *Proceedings of Supercomputing '90* (November 1990).