Code Reordering and Speculation Support for Dynamic Optimization Systems

Erik M. Nystrom, Ronald D. Barnes, Matthew C. Merten, Wen-mei W. Hwu
Center for Reliable and High-Performance Computing
University of Illinois
Urbana-Champaign, IL 61801
{nystrom,rdbarnes,merten, hwu}@crhc.uiuc.edu

Abstract

For dynamic optimization systems, success is limited by two difficult problems arising from instruction reordering. Following optimization within and across basic block boundaries, both the ordering of exceptions and the observed processor register contents at each exception point must be consistent with the original code. While compilers traditionally utilize global data-flow analysis to determine which registers require preservation, this analysis is often infeasible in dynamic optimization systems due to both strict time/space constraints and incomplete code discovery.

This paper presents an approach called Precise Speculation that addresses these problems. The proposed mechanism is a component of our vision for Run-time Optimization ARchitecture, or ROAR, to support aggressive dynamic optimization of programs. It utilizes a hardware mechanism to automatically recover the precise register states when a deferred exception is reported, utilizing the original unoptimized code to perform all recovery. We observe that Precise Speculation enables a dynamic optimization system to achieve a large performance gain over aggressively optimized base code, while preserving precise exceptions. For an 8-issue EPIC processor, the dynamic optimizer achieves between 3.6% and 57% speedup over a full-strength optimizing compiler that employs profile-guided optimization.

1 Introduction

Effective code optimization and scheduling requires register liveness information at all control-flow instructions. Instructions cannot be safely hoisted above conditional branches if their destination registers are live along the taken path from the branch Unfortunately, the cost of performing data-flow analysis to determine liveness at run-time is often great and can be exacerbated by the use exception handling mechanisms to support trapping and recovery of unexpected or error conditions. Some programs require *precise exceptions*, where both the ordering of exceptions and the processor and memory state at the time of the exception must be preserved. The additional control flow arcs that must be considered for the preservation of these precise exceptions render run-time data-flow analysis prohibitively expensive.

In order to alleviate some of the restrictiveness of supporting precise exceptions over the whole program, compilers often place restrictions on what an exception handler can expect about the region it protects. For example, the compiler may limit reordering of PEIs to within a single protection region. This assumption requires the programmer to specify separate protection regions around all order-dependent potentially excepting instructions (*PEIs*), but also allows for more aggressive optimization of larger regions. If a run-time optimizer is not free to make such assumptions, it is forced to maintain precise exceptions. Optimizers which reorder instructions must take precautions to preserve ordering and context at all PEIs. This can severely limit the aggressiveness of the optimizations performed because a large number of instructions in the dynamic instruction stream can except, including loads, stores, and float-point arithmetic.

Consider the code sequence in Figure 1a. A compiler with a view of an entire function may be able to perform data-flow analysis to determine the liveness of registers along branching control flow paths. For example, `r7` is the only register live out of branch `I`. Compilers may also register allocate across exception paths allowing registers to be live into exception handlers, such as `r4` out of excepting load `C`. Because of space and time constraints, post-link optimizers generally cannot perform detailed data-flow analysis and therefore cannot determine which registers are live out such paths. Figure 1b depicts the live-out sets for the same region of code when data-flow analysis is only performed on the trace. Because it is not known whether `r5` is live-out along the exception arc from load `D`, the redefinition of `r5` in add `E` cannot be moved above load `D`. Likewise, add `E` cannot be moved below branch `G` because it may be

Original Code	Original Live–Out
`A: branch` →	None
`B: r4 =`	
`C: load r2 =` →	r4
`D: load r3 =` →	r4,r2
`E: r5 = r3 + 1`	
`F: r6 = r2 + 4`	
`G: branch` →	r4,r6
`H: r7 = r4 + r5`	
`I: branch` →	r7

(a)

Trace Code	Assumed Live–Out
`A: branch` →	orig(r4,r2,r3,r5,r6,r7)
`B: r4 =`	
`C: load r2 =` →	r4,orig(r2,r3,r5,r6,r7)
`D: load r3 =` →	r4,r2,orig(r3,r5,r6,r7)
`E: r5 = r3 + 1`	
`F: r6 = r2 + 4`	
`G: branch` →	r4,r2,r3,r5,r6,orig(r7)
`H: r7 = r4 + r5`	
`I: branch` →	r4,r2,r3,r5,r6,r7

(b)

Figure 1. Live-out variables with respect to branches and potentially excepting instructions.

live-out along the branch's taken path. Essentially, add E is pinned between load D and branch G.

This paper presents a new model for speculation, called *Precise Speculation*, that automatically uses shadow registers to speculatively perform operations, while allowing quick restoration of precise state at the point of any taken branch or exception. Since precise architectural state is maintained at every non-speculative branch and PEI, this model improves upon mechanisms where entire traces are speculated. Rather than losing all performed work upon exit from the trace, the proposed method preserves all of the non-speculative computation that has been performed.

The relationship of Precise Speculation with previous methods of speculation is described in Section 2, and a complete explanation of this new method is given in Section 3. Scheduling results enabled by the proposed method are presented for multiple benchmarks in Section 5.

2 Related Work

A number of speculation techniques have been proposed to ensure correct exception handling and instruction reordering. Hardware out-of-order mechanisms maintain the appearance of sequential execution by internally buffering speculative changes to architectural state at an instruction, or even whole trace, granularity. In addition, a number of compiler techniques have also been proposed that allow the compiler to aggressively reorder instructions by utilizing architectural features to temporarily defer exceptions. While some accurately preserve precise exceptions, others may not preserve exception ordering or may drop some exceptions completely. Clearly, the compiler has an advantage over a dynamic optimizer because its optimization phase can perform analysis, generate code, and allocate registers for both the application and the exception handlers.

2.1 Hardware Speculation Mechanisms

Some out-of-order execution processors preserve precise exceptions by deferring the commits of speculative instructions. Memory and register modifications are buffered in their program order into a retirement structure called a *reorder buffer* [21]. Instructions in the buffer are not allowed to affect state until all older instructions have completed and committed. Checkpoint-repair mechanisms [12] have also been proposed to periodically preserve state. At checkpoints, copies of the register file are made, while between checkpoints, lists of changes to memory are recorded. Upon an exception, the register file copies are used to restore the previous register state while the lists of memory changes are used to undo each memory change. The size of the reorder buffer and difference lists limits the amount and distance of reordering that can take place when using these mechanisms.

Transmeta's Crusoe processors [6], Trace Processors [19], and IBM's Binary-Translation Optimized Architecture (BOA) [10] employ methods by which entire traces are speculated, committing the results of the traces only upon their successful completion. Any speculation failure within a trace forces recovery back to the beginning of the trace, and re-execution in the original code. This technique requires that all register writes occur to a shadow register file that only commits its values into the architectural register file upon successful completion of the trace. Likewise, a gated store buffer [23] prevents stores from being committed to memory until the end of the trace.

The Frame Cache [17] uses essentially the same trace speculation technique as the Crusoe processors, but reduces the number of trace side exits by converting unlikely side exits into *asserts* [18]. An assert is an instruction that triggers a trace speculation failure when its computed condition is false. When an unlikely side exit is taken (assert failure), the speculative state is thrown away and execution resumes at the beginning in unoptimized code. Use of asserts increases the size of an atomically executing unit from a basic block to an entire trace, whereby reducing the load on the branch predictor and increasing the scope of dynamic optimization. However, when an assert late in the trace triggers, all of the results of the speculated trace must be thrown away and execution restarted. Not even the successfully executed portions can be retained.

2.2 Compiler-Controlled Speculation Mechanisms

Early Very Long Instruction Word (VLIW) architecture work revealed that preserving precise exceptions severely limited the aggressiveness of optimization that could be applied at compile time. Multiflow Computer's TRACE architecture [7] utilized non-excepting forms of certain instructions, often called *silent* instructions, to allow for code movement above branches and other excepting instructions. When these instructions would except, a garbage value would be written into the result register. This phony result would sometimes cause future invalid computations and exceptions, but clearly these exceptions would not be precise. This form of compiler-controlled speculation is sometimes referred to as *general speculation* [4]. In such compiler-controlled speculative mechanisms, live register analysis can be performed to locate architectural registers that can hold the speculative values.

Boosting is another compiler-controlled mechanisms that is able to speculate arbitrary instructions [22]. In this model, each speculated instruction is tagged with the proper execution path to its home block. If the home block is reached along the specified path, the speculative results commit, otherwise they are cleared. This mechanism requires bits in the instruction opcode to represent directions of a conditional branch along the proper path to its home block. If a boosted instruction causes an exception the program counter (PC) is written into the instruction's result register, and the exception is delayed until the speculation is resolved. The PC value is propagated through the remaining boosted instructions into their result registers. If the path stored in the original instruction matches the actual executed path and the incoming operand is tagged as excepting, a speculated exception has occurred. In order to be precise, the program counter is used to find the offending instruction, and all subsequent, boosted instructions are re-executed non-speculatively. This ensures that recoverable exceptions, such as page faults, are correctly handled. This system requires both a shadow register file and a shadow store buffer to hold the boosted results.

Sentinel scheduling is a compiler technique designed to allow boosting of instructions above a guarding branch while accurately detecting and reporting all exceptions [15]. Instructions that are boosted above a branch are marked as speculative. When a speculated instruction causes an exception, the destination register is marked as having excepted, but no exception is reported. The program counter of the excepting instruction is instead stored in the register, which is propagated through subsequent speculative instructions, as with Boosting. Speculated instructions are required to have a check or non-speculative consumer instruction in their original location to detect and report the exception. When a non-speculated or check instruction reads an input register marked as excepting, the exception is reported with the program counter that is located in the source register. However, when multiple input registers are marked with excepting results, the exception propagated is arbitrary, thus not preserving precise exception ordering. The sentinel model also requires the use of extra recovery code to correct speculation failures due to recoverable faults. No extra hardware is required to support this model other than the excepting bits on the registers, and the support for propagating the program counter through speculative instructions. Inline recovery [1] utilizes sentinel scheduling but allows for re-execution of code between the speculated instruction and the check to serve as the recovery code.

The Boosting and Sentinel speculation models are not ideal for run-time optimization. Boosting requires extra opcode bits to represent the valid path, a feature that is not present in most architectures. Sentinel speculation does not preserve precise exceptions and would require rigorous data-flow capabilities for finding free registers in which to store speculated values. Finally, both models require recovery code, inline or explicit, to re-execute the instructions in order to service recoverable faults. In post-link reoptimization systems, recovery code may be difficult or inconvenient to generate. Since, these systems have access to the original code which is in pre-optimization program order, the original code can be used directly for recovery.

The DAISY dynamic translation system [20] utilizes a speculation mechanism similar to sentinel speculation. In this system, extra registers are available to the translation system that are not utilized by the native application. Speculative results can be written into these registers without affecting committed program state. Then, the values can be copied from the extra registers to architected registers inside the instruction's home block. Like sentinel speculation, an excepting instruction writes its program counter value into the result register and the value is then propagated through other speculated instructions. When an exception is reported through a non-speculative instruction, an exception to the DAISY system is triggered, which initiates the process of finding the offending instruction in the original code. Finding this instruction is necessary because its program counter value must be reported to the application exception handler to ensure precise exception handling.

Translations of original code to DAISY architecture code occurs on a page-by-page basis, which provides a unique method for finding the original instruction. The system locates the translated instructions for page entry points in the original architecture through a direct mapping to a location in the translated code. Therefore, correspondence between the original and translated entry points can be achieved through a simple calculation. When an exception occurs in the translated code, a backward path to an entry point through the translated code is found and recorded. This

path is then used to trace through the original code to find the offending pre-translation instruction. While this system correctly reports the program counter of the excepting instruction, DAISY requires significant overhead to locate the paths. Furthermore, DAISY must also be able to re-shuffle significant portions of translated pages. When a page is translated, the known entry points are placed at their mapped locations within the page, while other instructions are placed where convenient. However, as new entry points are found through subsequent execution, their mapped locations may be occupied with other instructions which must be moved in order to correctly translate the new entry points.

Repair mechanism techniques have been proposed for performing optimization while preserving precise exceptions in dynamic translators [14] [9]. In these methods, just enough processor state is maintained to allow repair code to be generated when an unexpected event, such as an exception, occurs. This repair code is executed to restore precise state before control transfers to any exception handler. However, due to the overhead of the generation of this repair code, such unexpected events must be very infrequent to prevent a negative impact on performance.

2.3 Other Dynamic Optimization Systems

The Dynamo dynamic optimization system is designed to transparently optimize a native application as it executes on the processor [2]. This system does not currently employ a speculation mechanism, but rather turns off any optimization that might violate precise exceptions when they are required. In Dynamo, instructions are initially interpreted while the instructions are profiled. *Fragments*, or traces, are formed along the frequently executed paths and then optimized. Since each fragment is constructed as a collection of basic blocks, the start, end, and all side exits of the fragment have consistent architectural state with their counterparts in the original program. Furthermore, in order to preserve precise exceptions, architectural state for all PEIs must also be consistent with their original counterparts. Because many types of optimizations performed on the fragments may remove instructions or move them with respect to PEIs, they can only be performed when the user indicates that precise exceptions are unnecessary for the application. Future work in Dynamo includes the use of optimized code debugging techniques during fragment construction in an attempt to allow for the reconstruction of the precise context from any point within the optimized fragments [24]. Trace optimizations within Dynamo would likely benefit from a hardware mechanism that assisted in preservation of precise exceptions.

In a similar way, hardware mechanisms [16] have been proposed to perform transparent profiling and trace formation in a memory-based code cache. Without performing data-flow analysis hardware systems cannot utilize sentinel speculation to allow rescheduling of instructions. Thus, in order to maintain precise exceptions, no instruction reordering can be performed. For more powerful optimizations to be performed, a more flexible speculation model is needed.

3 Precise Speculation

For greater run-time optimizer effectiveness, it is desirable to aggressively optimize across PEIs that are unlikely to except, while maintaining a mechanism for restoring precise exception state when an exception actually occurs. In most applications, exceptions are truly exceptional events. While these applications contain a large number of PEIs, many applications never actually except. Other applications may use exception handling as a convenient method for executing control flow changes, and so a number of potentially excepting instructions may except somewhat frequently. When optimizing code in the presence of PEIs it is desirable to optimize across the PEIs that are unlikely to except at the expense of recovery time from the exception. For PEIs that are more likely to except it is instead profitable to preserve quick recovery.

3.1 Sentinel Speculation - Accurate but Imprecise Exception Reporting

As described in Section 2, Sentinel Speculation allows potentially excepting instructions and their dependents to be moved above conditional branches and predicate defines in the code schedule while reporting only the appropriate, non-spurious exceptions. Consider the original code as shown in Figure 2a. In order to hide some of the latency of load instructions `C` and `D`, the scheduler moves them earlier, in this case, above branch `A`, making `C` and `D` speculative instructions. Figure 2b depicts the result of the code movement. Loads `C` and `D` have been converted into speculated loads, as denoted by the `S` flag prior to the opcode. In the Sentinel Speculation model, these instructions do not report their exception immediately, but rather record the exception condition in their result registers. Generally, a non-speculative check instruction is placed in the original location of the instructions to signal the actual exception. In order to limit the code size expansion, implicit checks can be used. With implicit checks, a non-speculative consumption of a register marked as excepting will signal the exception. For example, add operations `E` and `F` utilize the load results non-speculatively, and will signal the exceptions.

Since any exception caused by `C` and `D` will be reported only when their results are used by `E` and `F`, an exception will be reported if and only if one would have been reported in the original program. Despite this fact, the new code in Figure 2b violates two precise exception requirements. First, should both loads except, it will report `D` before `C` be-

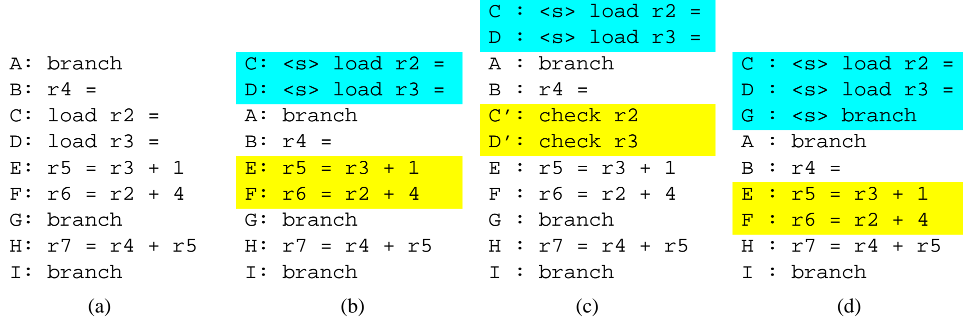


Figure 2. Code motion using Sentinel speculation and Precise speculation.

cause the ordering of the uses does not match that of the original loads. We will refer to this as the *ordering* requirement of precise exception reporting. Explicit checks could be inserted, as shown in Figure 2c, to preserve the detection order of exceptions at the cost of expanded code size. Obviously, a scheme that preserves the order of exception reporting without explicit checks is desirable. Our proposed new model is one such scheme.

Second, in Figure 2b, should only `C` except, the exception will not be known until the first use of `r2` at `F`, by which time `E` will have changed the value of `r5`. However, in the original code, the register write of `r5` should not have occurred before `C`'s exception is reported. This can potentially cause a problem with an exception handler if the old contents of `r5` is needed during exception handling. We will refer to this requirement as the *liveness* requirement of precise exception handling. This term reflects the fact that some registers can be used before being redefined in an exception handler.

In order to accurately conform to this requirement without special hardware support, the optimizer must construct a complete control flow graph including the exception handler code and perform global live register analysis, a form of global data flow analysis. Unfortunately, most proposed post-link optimization mechanisms either do not have guaranteed access to the entire control flow graph due to incomplete instruction discovery or cannot afford to perform global data flow analysis due to limited time/space budgets for their operation. As a result, very little reordering could safely take place across PEIs and branches, inhibiting almost all profitable code motion.

3.2 Overview of Proposed Model

To enable aggressive rescheduling and other dynamic optimizations, a new speculation model, called *Precise Speculation*, is proposed to minimize the restrictions placed upon the code reordering activities performed by the optimizer. The code reordering activities covered by the Precise Speculation model include standard control and data speculation across PEIs (though data speculation is left for a future work), as well as speculative relocation of PEIs and branches while completely preserving precise exceptions. This new speculation model is designed so that it could be utilized by hardware-only run-time optimizers, as well as by software-only systems like Dynamo [2] and software-driven hardware coprocessor optimizers [5] where complete data flow information is usually either too costly or infeasible to derive. Precise speculation adheres to both the *ordering* and the *liveness* requirements for maintaining precise exceptions.

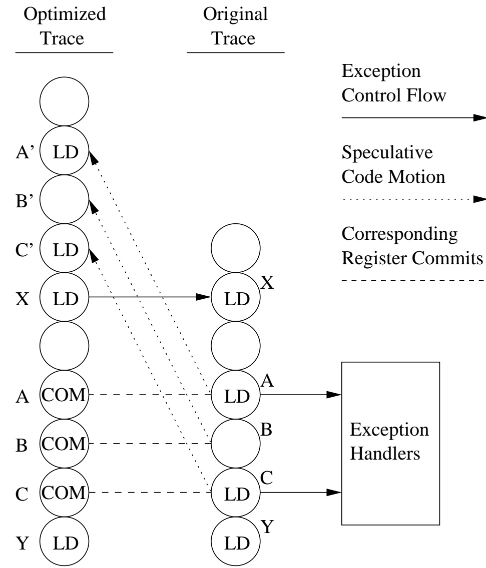


Figure 3. Precise Speculation concept.

Under the Precise Speculation model, PEIs and branches are for all practical purposes identical. Both PEIs and branches can change control flow to the point of a non-sequential instruction (branches to their branch target and exceptions to the exception handler). The only difference is that PEIs are generally assumed to branch much less frequently, thus are implicitly predicted as not-taken. Branching Instructions (BIs) will be used in reference to the set of all PEIs and branches.

Figure 3 depicts a high-level example of how Precise

svalid	Spec. Read	Spec. Write	Non-Spec. Read	Non-Spec. Write
0	R-Spec	W-Spec svalid=1	R-Spec	W-Both
1	R-Spec	W-Spec	R-Spec svalid=0 non-spec=spec	W-Both svalid=0

Table 1. Actions for register file accesses.

Speculation works. Consider the original trace of instructions shown on the right side of the figure, which consists of a set of potentially excepting loads and several other arbitrary instructions. In the example, the scheduler moves instructions `A`, `B`, and `C` up above load `X` as shown by `A'`, `B'`, and `C'` in the optimized trace on the left side of the figure. However, these speculative instructions must not destroy the contents of registers that may be live out of branch `X`. The proposed mechanism provides a separate speculative register file. Speculative instructions write their results into the speculative register file. At the proper time, the speculative results must then be committed into their proper destinations within the architectural registers. This is shown in the optimized trace as register commit instructions `A`, `B`, and `C`. Non-speculative instructions, on the other hand, write directly into their architectural destination registers.

With precise speculation, non-speculative BIs serve as checks for speculative instructions. For example, in Figure 3, non-speculative load X is the BI that serves as the check for `A'`, `B'`, and `C'`. When an instruction acting as an implicit check detects any speculation exceptions, the contents of the speculative register file is discarded. This removes the effects of all the speculative instructions whose results have not been committed. In Figure 3, the result of `A'`, `B'`, and `C'` will be discarded when any of them causes an exception that is detected by `X`. This ensures that all live registers are properly protected against any changes by the speculative instructions. A transition to original code is then forced. In Figure 3, the transition to original code corresponds to the transition of the execution from `X` on the left hand side to `X` on the right hand side. Since the results of `A'`, `B'`, and `C'` are all excluded from the architectural register file, the original code can correctly execute in their original order. Then, exceptions are reproduced in their proper order. It should be noted that the ordering of stores is never changed and that stores are never speculated.

3.3 Speculative Register File

Precise Speculation requires a special register file and works in tandem with a trace optimization system to allow exception recovery. Conceptually, each register contains a speculative half and non-speculative half as well as bit (*svalid*) that specifies whether or not the speculative half is valid. The effects of accesses to the register file are shown in Table 1. Non-speculative writes always write both halves of the register and clear svalid. Speculative instructions write only the speculative half and set svalid. All reads from a register read from the speculative half. However, non-speculative reads additionally clear the svalid bit and, if svalid is set, commit the outstanding speculative value by copying the speculative half to the non-speculative half. Speculative reads leave the svalid bit unchanged. Explicit commit instructions are no-ops composed of the sources ready for committing. The register file can be restored to a non-speculative state by copying the non-speculative half into the speculative half for each register with a set svalid bit.

When building the optimized trace for the code cache, some BIs may be speculated and some may be left unspeculated. Speculative BIs except silently, meaning that they flag an exception but otherwise do nothing. Non-speculative BIs except if either the instruction itself causes an exception or if a preceding speculative BI silently excepted. When a non-speculative BI excepts, it restores the register file to a non-speculative state as specified above and transfers control to a point just before its counterpart instruction in original code. The mapping between the optimized and original code is known at all non-speculative BIs and is described in Section 4. Once back in the original code, execution continues at a point prior to any of the speculated instructions, in effect executing inline recovery code. If the exception in the optimized code is recoverable, such as a deferred page fault, it will be serviced in the original code. After returning from the exception handler, execution continues normally and will eventually transition back to optimized code at an entry point.

Basic blocks should be viewed as regions between all BIs instead of just branches. Precise Speculation allows all instructions (including BIs themselves) to be safely speculated up into another basic block. The speculation of a BI allows the computational part and excepting part to complete early and allows the basic block above and below the BI to be merged into one, larger block. The commit for a BI can safely and implicitly occur at the first non-speculative use of the destination, or if none exists, anywhere in the new basic block using an explicit commit. In this manner, Precise Speculation can provide greater freedom in code motion by enlarging basic blocks.

3.4 Branch Speculation

The code in Figure 2b, while not sufficient to allow precise exception handling for sentinel speculation, is perfectly valid in precise speculation. In the figure, `A`, `C`, and `D` are all BIs. Should either `C` or `D` silently except, `A` will restore the register file and branch to just before its counterpart in original code. Assuming branch `A` in original code does not take (it could have been a guarding branch to make sure that `C` and `D` do not load from NULL), the loads will then be re-

executed and the exception handled as if the optimized code never existed. Previously, it was noted that branches and PEIs are considered in an identical manner. This means that branches can also be speculated under this model. Figure 2d shows an extension of the example in Figure 2b where a branch, along with the loads `C` and `D` is speculated. The speculated branch `G` silently excepts if its branch condition is true. This means that if the branch should be taken, the next non-speculative BI, branch `A`, acts the same as if one of the speculated loads had excepted. By speculated branch `G`, the basic block has been further increased in size.

The ability to speculate branches is presented to provide insight into the nature of precise speculation. However, this feature is not taken advantage of for the experiments.

3.5 Combining Sentinel and Precise Speculation

As discussed previously, PEIs (except for stores) can be precisely speculated above other non-speculative PEIs. The non-speculative PEIs serve as the exception check of all outstanding precise speculative instructions, forcing a branch back to original code. Back in original code, execution resumes non-speculatively in original program order. This execution guarantees the proper ordering and register context of exceptions.

While instructions that actually except can be included and optimized within a trace, care must be taken in choosing how far to speculate instructions that normally do not except. When an instruction is precise speculated to an earlier location within a trace, it may except more often if it is moved before a guarding branch. This exception would force an early exit from the optimized trace at the next non-speculative PEI, when the guarding branch would have normally protected it. It is beneficial, however, to precisely speculate instructions above branches that are highly likely to fall through. Because these branches are not likely to exit out of the trace, they are also not likely to be an important guard instruction for the PEI. Therefore precise speculation above branches with a strong fall through bias will limit likelihood of a premature exit.

Much of the benefit from compiler-generated sentinel speculation is that loads can be moved up above guarding branches. Since the loads are silent, the exception is deferred until a sentinel is reached which triggers the real exception. In the same way, a run-time optimizer could make potentially excepting operations both sentinel and precisely speculative by inserting an explicit sentinel for the instruction. As long as this sentinel is itself never speculated, spurious returns to unoptimized code can be avoided. However, since the goal of this work is to demonstrate the costs and benefits of our new model of speculation, we did not allow our optimizer to perform additional sentinel speculation.

Compiler-generated control-speculative instructions must have an eventual sentinel to prevent the loss of an exception. Since these sentinel speculated instructions are silent, a dynamic optimizer is free to precisely speculate them, even above branches with a good chance of taking.

3.6 Implementation Considerations

Although each register file entry in the precise speculation model is conceptually organized as a two-portion entity, most of the complexity can be confined to the retirement stage of the processor pipeline. In an implementation, one can simply have the speculative portion of the register file entries in the core data path. Thus the register file in the core data path will be referred to as the speculative register file, which is identical in design to the registers used in traditional architectures. An additional architectural register file with an svalid bit for each entry is provided in the instruction retirement stage of the processor pipeline.

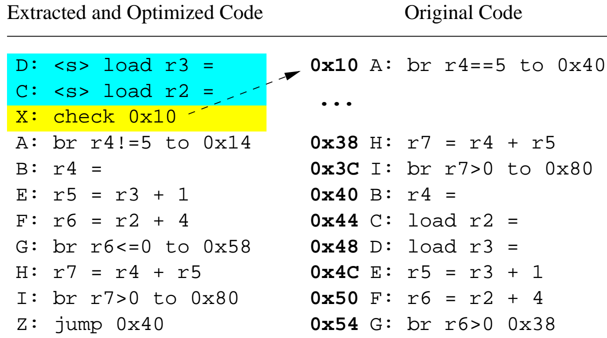
All instructions, speculative or non-speculative, write to the speculative register file. As non-speculative instructions are retired, they also write into the architectural file. On the other hand, speculative instructions are not allowed to write into the architecture file when they retire. They simply set the $svalid$ bit of their destination register entry. Thus, at any point in time, the svalid bits mark all the entries whose speculative and architectural portions differ. The value in the architecture file is the potentially live value being protected for precise exception.

As a non-speculative instruction retires, it also deposits its input register values into the architectural registers if these values were produced by speculative instructions. This effectively advances the contents of the architecture register file as the legitimacy of the speculative instructions are confirmed by their non-speculative dependent instructions. The $svalid$ bits can be used to avoid unnecessary copying of identical values from the speculative register file to the architectural register file. Additionally, buffering can be used to eliminate the need to increase the number of ports to the architecture file. Since the speculative instructions are not allowed to write to the architecture file, the buffered updates will eventually find idle ports into the file.

When a speculative exception is signaled by a check, either implicit or explicit, the speculative register file is repaired with the contents of the architecture file before the execution is redirected to the original code. This erases all the speculative instruction results and preserve all potentially live registers.

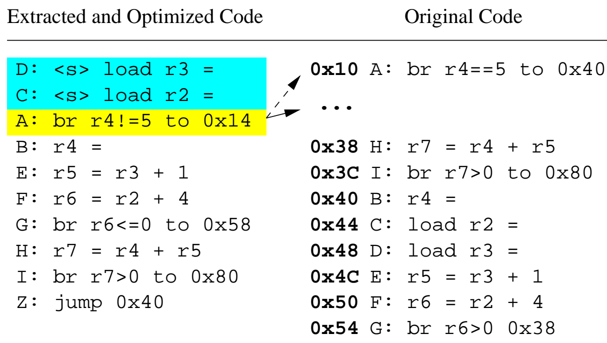
4 Mapping of Optimized to Original Code

In order to ensure precise exceptions, the original address of the excepting instruction must be reported to its handler. In many systems, the handler compares the instruction address to the handler's coverage range to determine if it covers the particular exception. In our proposed



(a) Explicit speculation check instruction

Hot execution path used during trace formation: A B C D E F G H I



(b) Implicit speculation check via non−speculated PEI

Figure 4. Explicit and implicit speculation check instructions.

system, the original address of the excepting instruction is also needed to begin inline speculation recovery back in the original code.

Figure 4(a) depicts a region of code that has been optimized in a dynamic optimizer. The right side of the figure depicts the original code region, and the left depicts the optimized code where loads C and D have been speculated and reordered. Using the precise speculation model, an explicit check instruction X has been inserted to direct execution back to the original code along the dashed line when an exception is detected. An explicit check instruction contains an address to its corresponding location back in the original code. When an explicit check is inserted, execution can immediately transition to the original code without delay, but incurs the cost of adding extra instructions into the code stream.

Figure 4(b) depicts the same region of code this time using an implicit, precise speculation check. In this mode, all potentially excepting and branching instructions also serve as checks. However, these instructions do not contain information as to their location in original code. Another mechanism must be utilized to obtain the address.

A number of solutions have been proposed to locate the original address. As previously described, DAISY walks backward in the optimized code tracking the path to an entry point where the original location is known. Other methods have included inserting markers into the code stream prior to each PEI with the corresponding original address. This technique is essentially equivalent to inserting an explicit check prior to each PEI. Finally, a table located outside of the optimized instruction stream could hold the corresponding addresses for each PEI.

Our solution is a hybrid scheme that allows the insertion of checks as address markers when explicit checking is desired or when an exception is likely while relying upon a table for the bulk of the markers. This way we gain the flexibility of a direct transfer to original code while minimizing the increase in code size.

Specifically, a table of original program addresses is created for each optimized trace, where each non-speculative PEI or branch in the code utilizes an entry. Instructions that cannot except do not serve as checks and therefore do not need entries in the table. Likewise, speculated instructions cannot branch and do not need entries, while any explicit check instructions already contain the original address and do not need entries. The challenge to any table-based method is finding the correct location in the table. In our method, we linearly allocate entries for the potentially branching instructions in the trace and use a counter to maintain the correct offset into the table. One additional consideration is the memory location of the table. Since the table is unlikely to be frequently accessed, it should be positioned such that it does not interfere with normal instruction caching operations.

At the start of a trace, the offset must be set to the start of the trace's table. To perform this initialization, a special instruction is inserted at the beginning of each trace, as shown as `initializer Y` in Figure 5. During execution of the trace, the counter is incremented for each PEI or branch that is encountered. When an exception is detected at a branching instruction, the counter will contain the appropriate index into the table. The processor discards any speculative values and transfers control to the target in the table pointed to by the offset.

Since branches can normally change control flow, there must be a precedence between its responsibility as an exception check and a conditional branch. Semantically, branch instructions compute their conditions *prior* to checking for exceptions. Should both a branch condition evaluate to true and an exception also be flagged, the branch will simply follow its target. This is valid because, effectively, code is only moved upward. There are two cases that illustrate this point. In the first case, the branch forces an exit from the trace back to original code. In the second case, the branch

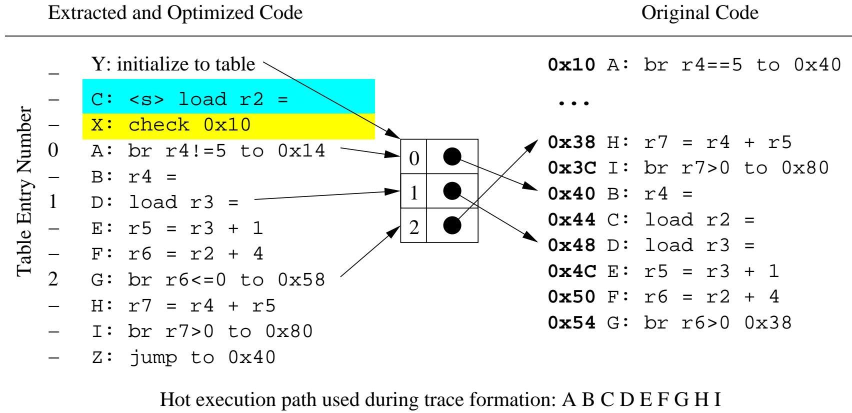


Figure 5. Hybrid approach to obtaining the original program address.

transfers control to another trace, or possibly to within the current trace serving as the loop back branch. In both cases, the speculative instruction that flagged the exception came from the branch's fall-through target and would not have been reached if executing in original order. This means that the flagged exception can be safely ignored.

Computing the condition of a branch and following its exit target first rather than checking for an exception permits two things. First, a taken branch never has to access the table to get to the original code because the target is encoded in the instruction. Second, the table offset for the branch can contain the fall through target of the branch instead of the address of the branch itself, as shown in `branch A` in Figure 5. This prevents the branch from having to re-execute should it see an exception flagged.

However, PEIs must execute first, and check for an exception second. Upon detecting an exception caused by a preceding speculative instruction or caused by the PEI itself, execution control must transfer to the original code just *prior* to execution of the PEI. For example, should a non-speculative load in the optimized code except, `load D` in Figure 5, the exception must be suppressed in the trace and re-execution of the instruction must be performed in the original code. This ensures that the program counter has the correct value at the exception and ensures that all speculative state has been discarded.

5 Experimental Analysis

To evaluate the potential benefit from Precise Speculation for run-time optimization, program hot spots [16] of several SPEC benchmarks were examined. These benchmarks were compiled using the IMPACT compiler [3] to generate aggressively-optimized superblock code [11] for the IMPACT EPIC architecture [1]. These programs were emulated and the Hot Spot Detector and Trace Generation Unit were simulated to extract sets of traces making up program hot spots [16].

Benchmarks	Num. Insts.	Inst from Selected Traces	Hot Spot Trace Sets	Static Ops	PEIs and Brs
099.go	329M	42M	7	13685	5066
124.m88ksim	80M	56M	3	3578	1395
129.compress	29M	12M	5	3787	1286
130.li	122M	63M	9	4701	2191
132.ijpeg	1.20B	704M	68	45093	17156
134.perl Jmble	2B	1.37B	8	5035	2672
134.perl Scrbl	2B	842M	4	5618	2923
181.mcf	509M	441M	9	4716	1714
183.equake	2B	1.26B	9	2940	852
197.parser	184M	78M	32	9414	4127

Table 2. Benchmark summary.

The benchmarks used for the experiments are listed in Table 2. Hot spots were detected using the method in [16]. Each hot spot consists of a set of traces, and we chose to only re-schedule and optimize hot spots containing at least one trace that executes more than 40,000 instructions through the course of the program. The number of instructions emulated is shown for each benchmark as well as the total dynamic number of instructions spent within the selected hot spots.

With the exception of one run of *perl*, traces were generated with the same input used by the static compiler to generate its profile information. This makes the comparison more challenging for the runtime optimizer. Results presented for the benchmark *parser* consist only of the dictionary read portion of its run while *perl* and *equake* were limited to 2 Billion instructions.

Each benchmark was first compiled to an internal representation, inlined, aggressively optimized, and scheduled for an eight issue IMPACT EPIC processor with three branch, three memory, three float-point, and five arithmetic units. Static compilation was guided by training profile information. Separate from the benchmarks, a subset of the standard C library was also compiled, with the aid of profile information for general usage of the library and with

inlining within the library itself. Instead of completing the compilation process to a binary, the benchmarks and the C library were left in IMPACT's *IR* (intermediate representation). The compiled IR was then emulated, hot spots were detected, and traces were formed. Once detected, the traces themselves were emulated. At the end of the emulation, selected traces along with their actual post-detection profiles were stored for subsequent experimental phases.

Performance results for the traces for each benchmark are presented and analyzed for the following four models:

1. *Original*: No changes made to the original code schedule. This model serves as a baseline for performance evaluation.

2. *Scheduled*: Rescheduling applied which preserves precise exceptions and limits early trace exits. As described in Section 3, any reordering of non-speculative instructions with respect to non-speculative PEIs requires that the moved instruction be precisely speculated. For our experiments, all commits are explicit and therefore added for every speculated instruction. Note that the speculation semantics require restricting each commit to the home block of the PEI it commits. Finally, to limit early trace exits, speculation does not proceed above unbiased branches which, for our experiments, is any trace branch whose taken threshold is more than 20%.

3. *Best scheduled*: Exceptions and early trace exits are assumed not to occur. The result is a loss of both precise exceptions and correct behavior on taken branches. Ignoring the potential for an early trace exit can cause the corruption of values live-out of a trace exit and the schedules produced are likely invalid. However, this is not a significant problem since these results are to serve only as a performance upper bound. By removing these two restrictions, no speculation overhead is required to re-order PEIs or instructions and PEIs, no commits are added, and instructions can be speculated higher since trace exit weights are not considered.

4. *Optimized*: Simple, exception-safe optimizations are performed, followed by the same scheduling process used for the *scheduled* model. Optimizations are limited to local copy and constant propagation, redundancy elimination and instruction rescheduling, all previously proposed for trace caches [8] [13].

Figure 6 shows the speedup results for the *Scheduled, Optimized, and Best Scheduled* cases. The *Original* is used as a baseline. Speedup on the traces range from 3.6% for *ijpeg* to about 57% for *perl (Jumble)*. The average speedup for rescheduled code bound to all of the constraints of the *Scheduled* model is about 22%. The speedup results from a variety of factors. During the trace generation process, traces are formed across functions both internal to the benchmark and through the included C library functions. This trace partial inlining greatly improves scheduling options. This is especially true with regard to the C library functions because a profile for the function for the specific benchmark did not exist at static compilation time. Additionally, the inlining is partial (i.e. it contains only the actively traversed portion of the function) allowing code to be inlined across functions not statically inlined due to code size limitations. Since permanent state is not changed by precisely speculated instructions until their matching commit is encountered, our model effectively allows the "violation" of registers live out of a branch. An instruction can be speculated above a PEI knowing that, should the PEI exit the trace, machine state will remain consistent.

The combination of partial inlining, separate library code, and the natural specialization during the dynamic optimization process can cause an individual trace profile to deviate from the aggregate profile used during static compilation. This is especially true when the benchmark input differs from the one used for the profile for static compilation, as is the case for *perl (Jumble)* in Figure 6. Perl was statically optimized using an average profile of the scrabble and primes inputs. The results labeled *perl_jm* are for a run using a different input named jumble while the results labeled *perl_scr* are those for a run using scrabble. For all three cases *scheduled, Optimized, and Best Scheduled*, rescheduling shows a larger speedup when the different input (jumble) was used. This provides evidence that a change in input can yield greater potential for re-optimization.

Another important point to extract from Figure 6 is the performance gap between the *Best Scheduled* and *Scheduled* cases. Part of the reason for the gap is that the scheduler has more freedom to perform upward code motion because it does not limit operations from passing an unbiased branch. Additional freedom comes from the fact that precise speculation requires the speculation of a PEI must cross at least one other PEI or it cannot be moved without the addition of an explicit check. To avoid inserting of such checks, in the *Scheduled* case we require that flow dependences must allow for an operation to move *at least* into another basic block for it to be moved at all. Speculation itself aggravates this because the basic block sizes grow as more speculation is performed increasing the distance over which a PEI will have to be relocated.

The other reason for the performance difference is the consumption of resources by the explicit commits added for every precisely speculative instruction. This effect is somewhat exaggerated since, in order to estimate a worst-case cost of the commits, we used only single source, explicit commits. Explicit commits tend to clump together since they cannot move outside of the basic block in which they

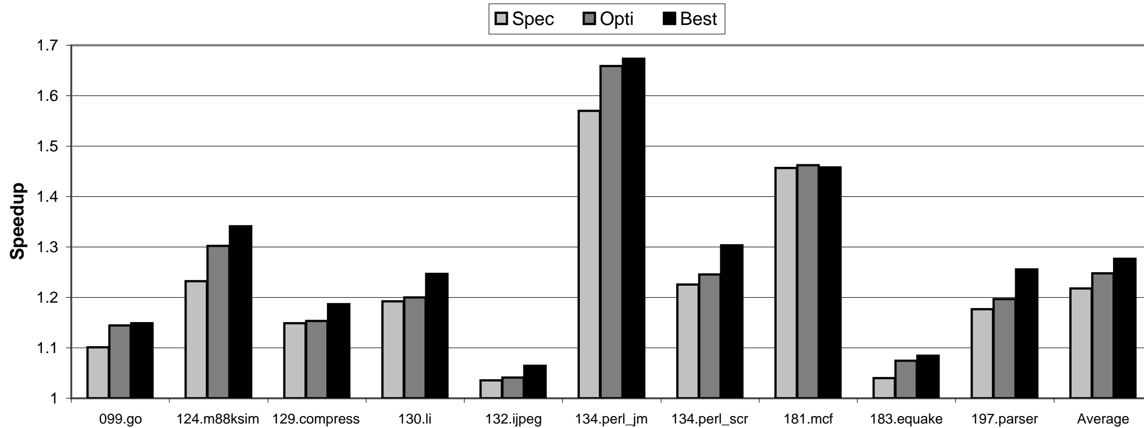


Figure 6. Speedup of traces under various scheduling constraints.

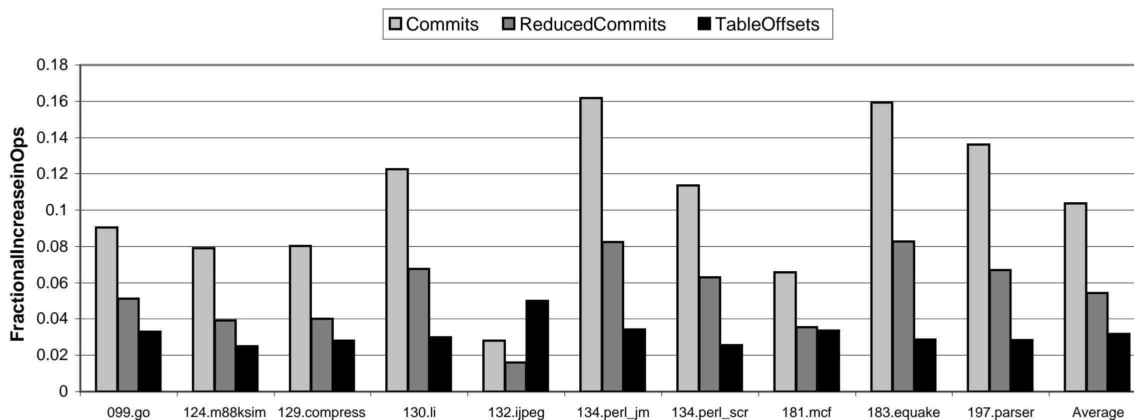


Figure 7. Fractional increase in code size due to commits and table offset instructions.

are placed. The fraction of ops that are commits for the *scheduled* model can be seen in Figure 7. Anywhere from 3% to 16% of all static trace instructions are commits. The *Reduced* column is the number of commits when taking advantage of implicit commits and two source explicit commits. This reduces the increase to between 1% and 8%.

The *Table Offsets* bar shows a count of the number of ops that must be added to keep the offset into original PC table consistent. One must occur are the beginning of each trace and one for a join at any loop-back branch internal to a trace.

6 Conclusion

The Precise Speculation mechanism allows dynamic optimization systems to perform aggressive code reordering and speculation while ensuring that exceptions are taken in their proper order. It further maintains a low-cost recovery mechanism to present precise register state when a deferred speculative exception is finally reported. The same recovery mechanism also eliminates the need for costly global live register analysis during global code motion. These problems have traditionally limited the effectiveness and usefulness of dynamic optimization systems. Precise Speculation solves these problems with a modest amount of hardware overhead that is mostly confined to the retirement stage of the processor pipeline.

We observe that supported by Precise Speculation, a dynamic optimization system can achieve a large performance gain over an aggressively optimized base code while preserving precise exceptions. For an 8-issue EPIC processor, the dynamic optimizer achieves between 3.6% and 57% speedup over a full-strength optimizing compiler that employs profile-guided optimization. All the observed performance benefits of the dynamic optimizer are derived from large and accurately formed optimization windows formed for each phase of the program execution. Without Precise Speculation, very little speed up would be possible since optimization and code reordering would be confined within instruction sequences between branches and potentially excepting instructions. In aggressively optimized code, there is virtually no opportunity for performance improvement in these limited instruction sequences.

Future work involves detailed studies of dynamic optimization and scheduling techniques that take full advantage of Precise Speculation, detailed characterization of each type of performance opportunity, and extending the model to support dynamic optimization of operating system code.

References

[1] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu. Integrated predicated and speculative execution in the IMPACT EPIC architecture. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 227–237, June 1998.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 1–12, June 2000.

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

[4] P. P. Chang, N. Warter, S. A. Mahlke, W. Y. Chen, and W. W. Hwu. Three architectural models for compiler-controlled speculative execution. *IEEE Transactions on Computers*, 44(4):481–494, April 1995.

[5] Y. Chou and J. P. Shen. Instruction path coprocessors. In *Proceedings 27nd Annual International Symposium on Computer Architecture*, pages 270–281, June 2000.

[6] R. F. Cmelik, D. R. Ditzel, E. J. Kelly, C. B. Hunter, D. A. Laird, M. J. Wing, and G. B. Zyner. *Combining hardware and software to provide an improved microprocessor*. United States Patent No. 6,031,992. Transmeta Corporation, February 2000.

[7] 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*, pages 180–192, April 1987.

[8] D. H. Friendly, S. J. Patel, and Y. N. Patt. Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors. In *Proceedings 31th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 173–181, December 1998.

[9] M. Gschwind and E. Altman. Optimization and precise exceptions in dynamic compilation. In *Proceedings of the 2000 Workshop on Binary Translation*, pages 66–74. ACM Computer Architecture News, March 2001.

[10] M. Gschwind, E. Altman, S. Sathaye, P. Ledak, and D. Appenzeller. Dynamic and transparent binary translation. *IEEE Computer*, pages 54–59, March 2000.

[11] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1):229–248, January 1993.

[12] W. W. Hwu and Y. N. Patt. Checkpoint repair for high performance out-of-order execution machines. In *Proceedings of the 14th International Symposium on Computer Architecture*, pages 18–26, June 1987.

[13] Q. Jacobson and J. E. Smith. Instruction pre-processing in trace processors. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, pages 125–129, January 1999.

[14] B. Le. An out-of-order execution technique for runtime binary translators. In *Proceedings of the 8th International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 151–158, October 1998.

[15] S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W. W. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel scheduling: A model for compiler-controlled speculative execution. *ACM Transactions on Computer Systems*, 11(4), November 1993.

[16] M. C. Merten, A. R. Trick, R. D. Barnes, E. M. Nystrom, C. N. George, J. C. Gyllenhaal, and W. W. Hwu. An architectural framework for runtime optimization. *to appear in IEEE Transactions on Computers Special Issue on Dynamic Optimization*, 2001.

[17] S. J. Patel and S. S. Lumetta. rePLay: A hardware framework for dynamic program optimization. Technical Report CRHC-99-16, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, December 1999.

[18] S. J. Patel, T. Tung, S. Bose, and M. Crum. Increasing the size of atomic instruction blocks by using control flow assertions. In *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 303–316, December 2000.

[19] E. Rotenberg, Y. S. Q. Jacobson, and J. E. Smith. Trace processors. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 138–148, December 1997.

[20] G. M. Silberman and K. Ebcioglu. An architectural framework for supporting heterogeneous instruction-set architectures. *IEEE Computer*, 26(6):39–56, June 1993.

[21] J. E. Smith and A. R. Pleszkun. Implementation of precise interrupts in pipelined processors. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 36–44, June 1985.

[22] 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*, pages 344–354, May 1990.

[23] M. J. Wing and G. P. D'Souza. *Gated store buffer for an advanced microprocessor*. United States Patent No. 6,011,908. Transmeta Corporation, January 2000.

[24] L.-C. Wu, R. Mirani, H. Patil, B. Olsen, and W. W. Hwu. A new framework for debugging globally optimized code. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 181–191, May 1999.