Unspeculation [*]

Noah Snavely Saumya Debray Gregory Andrews
Department of Computer Science
The University of Arizona
Tucson, AZ 85721, USA
Email: {`snavely, debray, greg`}`@cs.arizona.edu`

Abstract

Modern architectures, such as the Intel Itanium, support speculation, a hardware mechanism that allows the early execution of expensive operations—possibly even before it is known whether the results of the operation are needed. While such speculative execution can improve execution performance considerably, it requires a significant amount of complex support code to deal with and recover from speculation failures. This greatly complicates the tasks of understanding and re-engineering speculative code. This paper describes a technique for removing speculative instructions from optimized binary programs in a way that is guaranteed to preserve program semantics, thereby making the resulting "unspeculated" programs easier to understand and more amenable to re-engineering using traditional reverse engineering techniques.

1 Introduction

It is well known that processor speeds are growing faster than memory speeds, which means that the performance gap between the processors and memory is also growing steadily. One effect of this is that high-performance processors may be hamstrung because the memory system cannot deliver data as fast as the CPU would like. To alleviate this problem beyond what is possible using conventional instruction scheduling techniques, advanced architectures such as the Intel IA-64 (Itanium) have offered an innovative architectural feature called *speculation*. The idea is to allow (long-latency) instructions to be executed much earlier than would be possible in traditional architectures—possibly before it is even known whether the results of the computation will be used—in the hopes that initiating such expensive computations early will result in their results being available if and when they are needed. Judicious use of speculation can lead to significant improvements in performance [6]. However, speculation alters the structure of generated code in ways that do not reflect any logic in the original source, and it can significantly change the placement of instructions relative to unoptimized code. As a result, speculation tends to make low-level code obscure and difficult to understand, analyze, and reverse engineer. This can complicate the task of maintaining or understanding software for which the original source code is unavailable, e.g., pre-compiled libraries or applications distributed as executables.

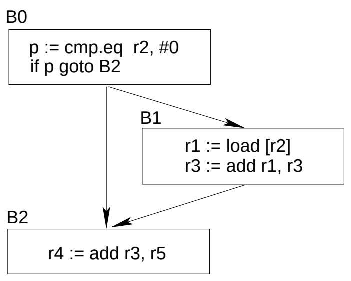
The contribution of this paper is to present a technique for undoing low-level optimizations based on speculation in order to expose the original structure of speculative programs and thereby to render them more amenable to the application of higher-level reverse engineering tools. We explain speculation in some detail, discuss how speculated code can be more difficult to understand than normal code, and describe a method for undoing optimizations based on speculation. The model for speculation we use follows that of the Intel Itanium, but the techniques we present are general enough to be applied to any architecture that supports similar speculative operations.

[*]This work was supported by the National Science Foundation under grants CCR-0073394, EIA-0080123, and CCR-0113633.

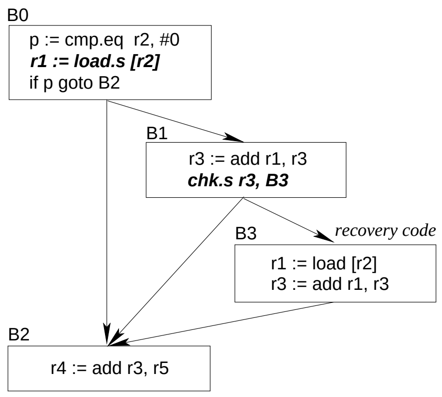
2 Background: Speculation

In order to generate efficient code, optimizing compilers attempt to hide the latencies of expensive operations by scheduling them as far apart as is necessary. However, instruction scheduling is constrained by dependencies between instructions. In particular, an instruction I that is *control dependent* on a conditional branch J—i.e., J determines whether I is executed—cannot, in general, be scheduled earlier than the branch instruction J. This is illustrated in Figure 1(a). Basic block B0 tests whether register r_2 contains a non-NULL value, and the load instruction in block B1 is control dependent on the branch in B0. Moving the load above the branch in this case would be incorrect: the resulting code would generate an error if r_2 has a NULL value. Such control dependencies limit our ability to hide the latencies of expensive operations such as loads from memory.

To address this problem, next-generation architectures, notably the Intel Itanium, have introduced an architectural feature called *control speculation*, whose essential feature is the speculative load instruction, denoted by the opcode '`load.s`.' The behavior of a speculative load is similar to that of a normal load, but with one important difference: If the instruction generates an exception, such as a segmentation or page fault, the exception is not handled immediately;



(a) Original unspeculated code



(b) Code with speculation

Figure 1: An example of control speculation

instead, a special bit associated with the destination register of the load, called a `NaT` ("Not a Thing") bit, is turned on. Later when the program reaches a point where the result of the load is needed, a special speculation check instruction (with the opcode '`chk.s`') is issued on the destination register of the load. If the register has its `NaT` bit set, then execution branches to recovery code provided by the compiler; otherwise, execution continues as normal. The `NaT` bits can propagate from one register to another. That is, if a source register of an instruction has its `NaT` bit set, then the `NaT` bit of its destination register will become set. This means that a string of dependent instructions can follow a speculative load, and in general these instructions will all have to be reissued in recovery code.

Using control speculation in the example shown above, we can move the load instruction above the preceding branch, in the process turning it into a speculative load. The resulting code,[1] shown in Figure 1(b), is considerably harder to understand than the original. First, there are more instructions, more execution paths, and more convoluted program structure to consider in the speculated code. Second, the speculative load has moved farther from its use, with intervening recovery code whose behavior has to be taken into account, thereby obscuring the original program logic. The problem is exacerbated even further in larger programs where the speculation is more aggressive—e.g., when a speculative load is moved across several conditional branches rather than the single branch in the example above—and where the recovery code may, for example, itself contain other speculative or check instructions, thereby resulting in significantly more convoluted control flow. The next section describes a method of unspeculating code that essentially reverses the process of speculation, and hence makes the code easier to understand.

3 Unspeculation

Unspeculation refers to the process of transforming a program containing speculative loads to a semantically equivalent program where some or all of the speculative instructions have been replaced by "ordinary" load operations. Our approach to unspeculation consists of two distinct phases. First, we move each speculative load to one or more points in the code stream where it can potentially be replaced by an unspeculative load operation. We call this *load sinking*. Second, we verify that the check and corresponding recovery code can safely be eliminated and hence that the speculative load can be replaced by an unspeculative load. Each of these steps must, of course, be semantics-preserving.

As an example, starting with the speculative code in Figure 1(b), load sinking involves moving the speculative load to the start of block B1. Recovery code verification involves checking that the results of executing blocks B1 and B3 are identical, and hence that the code in Figure 1(b) can be simplified to that in Figure 1(a). Both steps are in general much more complicated than illustrated by this example, because there is not necessarily a one-to-one correspondence between speculative loads and checks, because recovery code does not necessarily contain the same instructions as regular code, and because exceptions are handled differently for speculative and unspeculative loads.

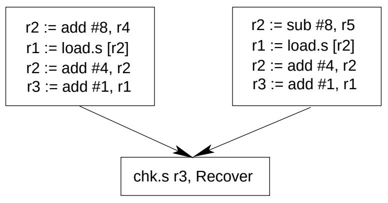
Below we describe in detail how we move speculative loads to be near check instructions, and how we verify whether recovery code and check instructions can be elim-

[1] For simplicity, we depart from the syntax of Itanium assembly instructions (which tend to be quite different from those of more familiar architectures) and write our instructions as follows:

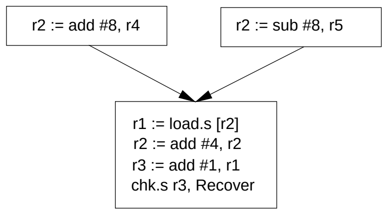
$$dst := op\ src_1\ src_2 \ldots$$

Here *op* denotes the operation, *dst* is the destination, and $src_1, src_2, \ldots$ are the source operands. A memory load instruction is expressed as a simple indirect access through a register, with any necessary address computations, displacements, etc., being carried out explicitly:

$$dst := load\ [r]\ .$$



(a) Before load sinking



(b) After load sinking

Figure 2: An example of load sinking

inated. Both analyses require examing possible instruction dependencies, which in turn requires determining whether memory addresses in registers might possibly overlap. Section 3.3 describes this memory-disambiguation problem.

3.1 Load Sinking

The main difference between "ordinary" and speculative load operations is that exceptions raised by the latter are deferred via the NaT bits. It follows that the appearance of a speculative load in a program indicates that it cannot be guaranteed to execute without any exceptions. In general, therefore, we cannot simply replace a speculative load by an unspeculative one and expect to preserve program semantics. Instead, the speculative load must be moved to some appropriate later point in the code stream.

The check instruction(s) associated with a speculative load indicates where a legal result for that load is expected, and hence suggests a natural placement for the load: immediately before the check instruction(s). In effect, this pushes the speculative load down into the basic block containing the corresponding check instruction, past any intervening conditional branches.

We refer to this process of moving speculative loads "down" towards their check instructions as *load sinking*. It is illustrated in Figure 2. Note that when a speculative load I is sunk, other instructions that depend on I must be sunk as well. To make this notion of "dependence" precise, define two instructions I and J to be *directly dependent* (written $I \rightleftharpoons J$) if:

1. I may write to any register or memory location that may be read by J; or
2. I may read from any register or memory location that may be written to by J; or
3. I and J may write to the same register or memory location.

Let $\rightleftharpoons^\star$ denote the reflexive transitive closure of the $\rightleftharpoons$ relation. We say that I and J are *dependent* if $I \rightleftharpoons^\star J$.

Load sinking is complicated by the fact that there may not be a one-to-one correspondence between speculative load and check instructions: a speculative load may be checked by several different check instructions, and a check instruction may check several different speculative loads. Moreover, in the latter case, the speculative loads may have different sets of dependent instructions. The various combinations of speculative loads and checks are illustrated in Figure 3. The remainder of this section addresses how to deal with these issues. The problem of memory disambiguation for identifying dependencies between memory reads and writes is discussed in Section 3.3.

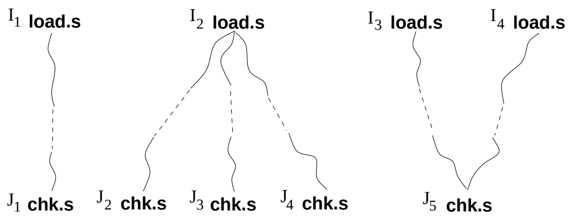


Figure 3: General structure of speculative computations

3.1.1 Finding relationships between instructions

Our first goal is to identify, for a given speculative load, the set of check instructions that test whether that load succeeded or failed. As mentioned in Section 2, however, a computation can propagate NaT bits from one register to another. For this reason, a speculation check associated with a speculative load into a register r may not check the register r itself, but possibly some other register r' whose value has been computed from that of r. This occurs in Figure 1(b), where the speculation check (in basic block B1) checks register r_3 even though the speculative load (in block B0) loads into register r_1.

In general, to determine whether a given check is associated with a given speculative load, we need to know whether or not the check's source register may be a NaT as a result of the failure of that load. To this end, given an instruction $I \equiv$ '`r := load.s ...`' that defines a register r and a check instruction $J \equiv$ '`chk.s` r', ...', we say that J *checks* I if either of the following hold:

1. $r' \equiv r$, and the definition I of r reaches J;[2] or

2. there is an instruction I' that uses r and that propagates NaT bits from its source operands to its destination, such that (i) the definition I of r reaches I', and (ii) J checks I'.

The set of speculation checks $Chk(I)$ associated with a speculative load I can then be defined as

$$Chk(I) \stackrel{\triangle}{=} \{J \mid J \text{ is a speculation check and } J \text{ checks } I\}.$$

In Figure 1(b), for example, since `add` instructions propagate NaT bits, the chain of reaching definitions along the execution path

```
r1 := load.s [r2]     # Block B0
r3 := add r1, r3      # Block B1
chk.s r3, B3          # Block B1
```

allows us to infer that the check instruction in block B1 is associated with the speculative load in block B0.

Given a speculative load I, the set $Chk(I)$ can be determined via a depth-first traversal of the control flow graph starting at I. At each point, we keep track of the set of *speculative registers* at that point, i.e., the registers whose NaT bits may be set. Initially, this contains only the destination register of the speculative load. It is updated during the traversal using information about instructions that propagate NaT bits. The traversal stops whenever the speculative register set becomes empty. The set $Chk(I)$ then consists of the speculation checks that can be reached in this traversal.

Analogous to the set $Chk(I)$ for a speculative load I, we can consider the set $Ld(J)$ of speculative loads associated with a check instruction J:

$$Ld(J) \stackrel{\triangle}{=} \{I \mid I \text{ is a speculative load and } J \in Chk(I)\}.$$

This set can be derived from the *Chk* sets computed for the speculative loads in the program.

3.1.2 Speculative regions

Intuitively, in order to carry out load sinking to a speculation check J, the set of instructions sunk to J must be well defined, i.e., must be the same for all speculative loads $I \in Ld(J)$. To see the reason for this, consider the speculative loads I_3 and I_4, and the speculation check J_5, in Figure 3. Let S_3 be the set of instructions dependent on the speculative load I_3, and S_4 the set dependent on I_4. When sinking I_3 we want to move all the instructions in S_3 down to the check instruction; when sinking I_4, similarly, we want to move all of S_4. If $S_3 \neq S_4$ it is not clear what instructions ought to be moved down to the check; if this happens, load sinking is said to fail.

[2]A definition I of a variable or register x is said to *reach* a program point p if there exists an execution path from I to p along which x is not redefined, i.e., along which the value assigned to x by I may survive [1].

To make these ideas precise, we define a speculative region as follows:

Definition 3.1 The *speculative region* of a speculative load I is a pair (L, C) where L is a set of speculative loads and C is a set of speculation checks, such that L and C are the smallest sets satisfying: (i) $I \in L$; (ii) if $x \in L$ and $y \in Chk(x)$ then $y \in C$; and (iii) if $x \in C$ and $y \in Ld(x)$ then $y \in L$. ■

A speculative region is unspeculated as a single unit. This means that for each such region, either load sinking succeeds and all speculative code in the region is moved at once, or it fails and no instructions are moved. To make this notion precise, consider an execution path π from a speculative load L to a check $C \in Chk(L)$. Let $Dep_L(\pi)$ denote the set of instructions along π that are dependent on L. We can now make precise the conditions under which load sinking can be carried out for a speculative region:

Definition 3.2 A speculative region (L, C) of a speculative load is said to be *path-independent* if, for any pair of speculative loads $I_1, I_2 \in L$ and check $J \in C$, and any two paths π_1 between I_1 and J and π_2 between I_2 and J, it is the case that $Dep_{L_1}(\pi_1) = Dep_{L_2}(\pi_2)$. ■

As an example, in Figure 3, path independence requires that the instructions dependent on the speculative load I_3 along the path $(I_3 \ldots J_5)$ be the same as the set of instructions dependent on the speculative load I_4 along the path $I_4 \ldots J_5$.

If a speculative region (L, C) is path-independent, load sinking becomes straightforward:

1. Let π be an arbitrary path from some load in L to some check in C and $S = Dep_L(\pi)$ the instructions on π dependent on L.

2. For each speculative load $I \in L$ delete the instructions S between I and any check in C.

3. For each check $J \in C$, copy the instructions S to the top of J's basic block. Additionally, if there are any non-speculative instructions S' in S that compute a value that is live along a path that leaves the region without going through a speculation check, copy S' onto this path.

The code structure resulting from load sinking is illustrated in Figure 4.

3.2 Recovery Code Verification

In Figure 4 there are two possible outcomes for the speculation check in block B_{chk}. If the speculative load completes successfully without setting any NaT bits, then execution takes the *pass path* $\pi_{pass} \equiv B_{chk} \rightarrow B_{fallthru} \rightarrow B_{merge}$. If the speculative load may fail and set NaT bits, then execution goes through the recovery code along the *fail path* $\pi_{fail} \equiv B_{chk} \rightarrow B_{rec} \rightarrow B_{merge}$.

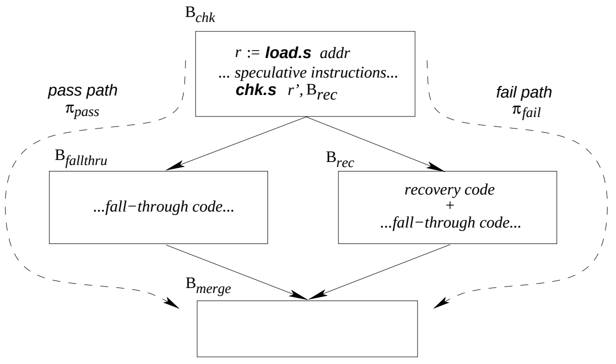


Figure 4: Code structure after load sinking

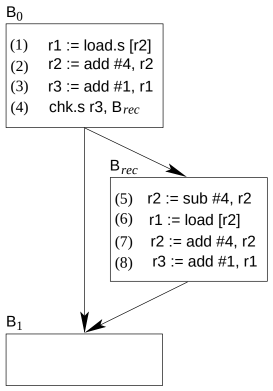


Figure 5: An Example of Recovery Code Verification

The effect of unspeculation is twofold. First, the speculation check instruction and the fail path π_{fail} are eliminated. Second, the speculative instructions in B_{spec} are converted to unspeculative ones, which means that exceptions deferred by the speculative code are no longer deferred after unspeculation. In order for this to be correct, the code must satisfy two conditions:

1. [*Path Equivalence.*] The execution paths π_{pass} and π_{fail} must be equivalent, in the sense that for every register and memory location x, the value of x at the entry to B_{merge} must be the same when execution goes along π_{pass} as when it goes along π_{fail}.

2. [*Load Equivalence.*] For every memory location y from which there is a speculative load in B_{chk}, there must be an unspeculative load from y in B_{rec}.

The need for the first criterion is obvious: if π_{pass} and π_{fail} can produce different values for some register or memory location, then eliminating π_{fail} in the course of unspeculation can potentially change the behavior of the program. The second criterion is motivated by the need to ensure that the exception behavior of the code after unspeculation is the same as that of the original code before unspeculation.

The remainder of this section discusses how we verify these criteria. Our current implementation is able to reason about path equivalence only when each of the pass path π_{pass} and the fail path π_{fail} is a single straight-line path with no branches. It can sometimes happen that the pass and/or fail path may contain other speculation checks that introduce branching structure into the code, but this gets eliminated during the course of unspeculation. To catch such situations, we iterate the unspeculation process until no more speculative code can be eliminated. As the experimental results reported in Section 4 indicate, this suffices for most instances of speculation encountered in practice.

3.2.1 Verifying Path Equivalence

The simplest case of path equivalence is when the recovery code is identical to the speculated code, except for the speculative load that is replaced with an unspeculative load. This occurs, for example, in the code in Figure 1(b). In general, however, the contents of registers may change between a speculative load through a register r and a check on that load, as illustrated in basic block B3 in Figure 2(b). To recover if the load fails, the correct address has to recomputed before reissuing the load, and so the recovery code needs extra instructions to fix the program state appropriately.

The general case is illustrated in Figure 5. Instructions 1-4 are from the longest block in Figure 2(b) after load sinking. Instructions 5-8 are the corresponding recovery block. The first instruction in the recovery code (instruction 5) undoes the changes to register r_2 after the speculative load, restoring its value to that at the speculative load. After this the load is reissued, this time unspeculatively. The remainder of the recovery code recomputes values that were computed using the result of the speculative load. As this example illustrates, both the speculative code and the recovery code may contain address and register computations, which have to be taken into account when reasoning about path equivalence.

Proving path equivalence involves reasoning about the contents of registers and memory locations along the pass and fail paths. In doing this, our current implementation is conservative in its treatment of memory: if either the pass path or the fail path contains any stores to memory among the instructions that are dependent on a speculative load, we conservatively assume that path equivalence does not hold, and abandon the unspeculation effort for that speculative region. This is not a significant problem in practice, as shown by our experimental results (see Section 4).

Given this treatment of memory stores, proving path equivalence boils down to reasoning about the contents of registers along the pass and fail paths. To do this, we specify a logical formula Φ asserting that there exist program states for which path equivalence does not hold—i.e., for some register r, the value of r along the pass path differs from its value along the fail path. We then use constraint solving techniques to try and show that Φ is unsatisfiable. If we are able to do so, we conclude that there are no program states that can cause path equivalence to be violated, and hence that path equivalence holds.

Given a logical formula A, let $(\exists)A$ denote the "existential closure" of A, i.e., the formula where all free variables in A are existentially quantified. Using this notation, we can write the formula Φ as:

$$\Phi = (\exists)[\ \Psi_p \wedge \Psi_f \wedge \Delta\ ]$$

where Ψ_p and Ψ_f are formulae expressing the values of locations at the end of the pass path and the fail path respectively, in terms of their initial values; and Δ is a formula stating that there is some location whose value at the end of the pass path is different from that at the end of the fail path, i.e., path equivalence does not hold.

Assume that each instruction in the program has a unique name I_k. We describe the construction of the formula Ψ_p, corresponding to the pass path, as a conjunction of the constraints specified below; the construction of Ψ_f, corresponding to the fail path, is exactly analogous. The value of a register r at the beginning and the end of the pass path are denoted by r_0^p and r_e^p respectively. At intermediate points along the pass path, the value of register r immediately after instruction I_k is denoted by r_k^p. For each instruction I_k along the pass path, Ψ_p contains a conjunct C_k that captures the effect of I_k. These are defined as follows:

1. $I_k \equiv$ 'r := `load` $[s]$'. In this case $C_k \ \equiv\ r_k^p = mem(s_j^p)$ where I_j is the most recent instruction that defines register s ($j = 0$ if s has not yet been defined along the pass path), and *mem* is an uninterpreted function symbol.

2. $I_k \equiv$ '$r := s \oplus t$' for some operation $\oplus$, and registers s and t, where the semantics of $\oplus$ is known to the analyzer. In this case, $C_k \ \equiv\ r_k^p = f_\oplus(s_i^p, t_j^p)$ where I_i and I_j refer to the most recent instructions defining registers s and t respectively; $i = 0$ (respectively, $j = 0$) if s (respectively, t) has not yet been defined along the pass path); and $f_\oplus$ expresses the semantics of the operation $\oplus$. Our analyzer knows about the semantics of some common arithmetic instructions: e.g., if $\oplus =$ `add` then $f_\oplus$ is the binary function '+,' signifying addition; if $\oplus =$ `sub` then $f_\oplus$ is '$-$,' signifying subtraction; etc.

3. Otherwise, the effects of instruction I_k cannot be modelled by the analyzer. The analysis is aborted in this case, and our system conservatively assumes that path equivalence does not hold.

Finally, for each register r, Ψ_p contains a conjunct expressing the final value of r. Let the last instruction along the pass path that defines r be I_k ($k = 0$ if r is not defined along the pass path), then this conjunct is given by

$$r_e^p = r_k^p.$$

As mentioned above, the construction of Ψ_f, corresponding to the fail path, is exactly analogous.

The formula Δ expresses that some register has a final value that is different along the pass and fail paths:

$$\Delta \ \equiv \bigvee_{r \text{ a register}} r_e^p \neq r_e^f.$$

In the actual implementation, we refine this process to reduce the size of constraints and the cost of checking satisfiability of constraints. First, it suffices to restrict our attention to the (usually small) set of registers that are actually modified along at least one of the pass and fail paths. Second, we reduce the number of instructions that we have to consider by walking backwards on each path from the merge point, marking instructions that are identical on both paths, until we reach two non-identical instructions or the top of the check block. If we happen to hit the top of the check block, then the relation becomes vacuously empty, so there is nothing to check. Our implementation uses the Omega calculator [11] to determine the satisfiability of the formula Φ.

The algorithm can be illustrated using the recovery code shown in Figure 5. We have $\Phi = (\exists)[\Psi_p \wedge \Psi_f \wedge \Delta]$, where:

$$\begin{aligned}
\Psi_p = \quad & r1_1^p = mem(r2_0) \\
& \wedge\, r2_2^p = r2_0 + 4 \\
& \wedge\, r3_3^p = r1_1^p + 1 \\
& \wedge\, r1_e^p = r1_1^p \\
& \wedge\, r2_e^p = r2_2^p \\
& \wedge\, r3_e^p = r3_3^p. \\
\Psi_f = \quad & r1_1^f = mem(r2_0) \\
& \wedge\, r2_2^f = r2_0 + 4 \\
& \wedge\, r2_5^f = r2_2^f - 4 \\
& \wedge\, r3_3^f = r1_1^f - 1 \\
& \wedge\, r1_6^f = mem(r2_5^f) \\
& \wedge\, r2_7^f = r2_5^f + 4 \\
& \wedge\, r3_8^f = r1_6^f + 1 \\
& \wedge\, r1_e^f = r1_6^f \\
& \wedge\, r2_e^f = r2_7^f \\
& \wedge\, r3_e^f = r3_8^f. \\
\Delta = \quad & r1_e^p \neq r1_e^f \vee r2_e^p \neq r2_e^f \vee r3_e^p \neq r3_e^f
\end{aligned}$$

The reader may verify that these constraints simplify in a straightforward way to give

$$r1_e^p = mem(r2_0) \wedge r2_e^p = r2_0 + 4 \wedge r3_e^p = mem(r2_0) + 1$$

$$r1_e^f = mem(r2_0) \wedge r2_e^f = r2_0 + 4 \wedge r3_e^f = mem(r2_0) + 1$$

whence the Δ constraints are falsified, which implies that Φ is unsatisfiable. This, in turn, implies path equivalence for the code in Figure 5.

3.2.2 Verifying Load Equivalence

Load equivalence can be determined using an approach very similar to that described above for path equivalence. The idea is to pair up speculative loads with unspeculative loads in the recovery code, and then to use a constraint-based test analogous to that above to determine whether the address registers being used in the two loads could have different values.

3.3 Memory Disambiguation

Memory disambiguation involves learning enough about the contents of registers at a given program point to decide if two registers can contain overlapping addresses at any time during execution. We need to solve this problem in order to determine whether instructions are dependent when doing load sinking and recovery code verification. The problem is difficult in general, and it is exacerbated here by the lack of semantic structure at the machine code level. Our current implementation generalizes a simple analysis technique known as *instruction inspection* [5]. The general idea here is that two memory references can be inferred to be non-conflicting if either (i) they use distinct offsets from the same base register r, with no intervening definitions of r; or (ii) they point to disjoint regions of memory, e.g., the stack and the global data area. The first of these is straightforward to adapt to the Itanium; due to space constraints we do not discuss it further. The remainder of this section focuses on the second technique.

We use a simple iterative dataflow analysis called *region analysis* to associate, with each memory reference in the program, a subset of the memory regions that the reference may access. The basic idea behind this analysis comes from the manner in which the different sections of an executable file are generated. The object module generated by a compiler from a source module typically consists of several code and data sections, e.g., the code section, the constant data section, the zero-initialized data section, etc. The linker combines a number of such object modules into an executable program: in the process, it puts all the sections in their final order and location. The sections of the same type coming from different object modules are typically combined into a single region of that type in the final executable. In general, when generating an object module from a source module, a compiler has no information about other object modules, e.g., their number, size, or the order in which they will be linked together, so it cannot make any assumptions about the eventual locations of these regions in the final executable. As a result, because the distance between the two regions of memory is not known at compile time, the code generated by a compiler for address computations cannot use a pointer to a particular region of memory to obtain an address pointing to some other region of memory. In other words, an address obtained by doing address arithmetic starting with a pointer to a particular region of memory can be safely assumed to fall within that same region of memory. This observation forms the basis of this analysis.

Our analysis considers the set of regions

$$\mathbf{D} = \{\mathsf{stack}, \mathsf{global}, \mathsf{GOT}, \mathsf{num}\}$$

where stack refers to stack locations, global to globals, GOT to the global offset table, and num to numerical constants. Here, the stack and global regions are self-explanatory. The global offset table is a read-only region of memory containing 64-bit addresses that are either code addresses or global data addresses.[3] Since memory disambiguation is relevant only when at least one of the references is a *store* instruction, and the text section (the memory region containing the actual executable code for the program) is read-only, we make the simplifying assumption that all addresses in the global offset table point to global data. This is safe, though in theory it may occasionally lose precision. The element num refers to numerical constants that may be computed as part of address computations.

Our analysis domain is the powerset of this set, $\mathcal{P}(\mathbf{D})$, ordered by subset inclusion; $(\mathcal{P}(\mathbf{D}), \subseteq)$ forms a complete lattice, with least element $\emptyset$ (denoting an unreachable reference), and greatest element $\mathbf{D}$ (denoting an unknown value), and meet operation $\cup$. Instructions within a basic block are handled as follows, with the notation '$r \mapsto S$' denoting that a register r points to a set of regions S:

1. If $r \mapsto S$, and the value of r' is obtained by an arithmetic computation involving r, then $r' \mapsto S$.

   This reflects the characteristics of address computations discussed above.

2. [*Standard register usage conventions*]: the register `r1` $\mapsto \{\mathsf{GOT}\}$; the stack pointer `sp` $\mapsto \{\mathsf{stack}\}$.

3. [*Loads from memory*]: Given an instruction

   $$r' := \mathsf{load}\ [r]$$

   if $r \mapsto \{\mathsf{GOT}\}$ immediately before this instruction, then immediately after this instruction we have $r' \mapsto \{\mathsf{global}\}$. Otherwise, if $r \mapsto S$ and $S \neq \mathsf{GOT}$, then $r' \mapsto \mathbf{D}$ after the instruction.

   This reflects the assumption above that addresses in the global offset table point to global data. We make no assumptions about the contents of other memory regions, so loads from them produce the value $\mathbf{D}$, denoting 'unknown.'

Set union is used as the meet operator to propagate information across basic blocks. Values are propagated iteratively until a fixpoint is attained, i.e., until there is no change to the set computed for any register.

[3]Other 64-bit architectures where the instruction width is smaller than 64 bits, e.g., the Compaq Alpha, use a similar table for handling 64-bit constants and absolute addresses.

Program	SPECULATIVE LOADS			SPECULATION CHECKS		
	Orig. (L_0)	Unspec. (L_1)	Reduction (%) ($(L_0 - L_1)/L_0$)	Orig. (C_0)	Unspec. (C_1)	Reduction (%) ($(C_0 - C_1)/C_0$)
bzip2	130	31	76.2	124	42	66.1
gzip	224	62	72.3	181	54	70.2
mcf	94	31	67.0	97	34	64.9
parser	483	85	82.4	451	75	83.4
twolf	1542	385	75.0	1399	354	74.7
vortex	5339	451	91.6	5217	352	93.2
vpr	608	152	65.0	614	145	76.4
GEOM. MEAN:			75.2			75.0

Table 1: Amount of speculated code before and after unspeculation

After the analysis, two indirect memory references through registers r_1 and r_2, where $r_1 \mapsto S_1$ and $r_2 \mapsto S_2$, can be inferred to be independent if $S_1 \cap S_2 = \emptyset$.

3.4 Putting it All Together

The discussion of unspeculation thus far can be summarized as the following sequence of steps:

1. Group the speculative loads and speculation checks into speculative regions (Section 3.1.2).
2. For each speculative region, verify path independence (use the memory disambiguation techniques discussed in Section 3.3 to identify dependencies between memory accesses). If path independence cannot be verified for a region, abandon unspeculation for that region.
3. For each speculative region that is path independent, carry out load sinking (Section 3.1).
4. Verify path equivalence and load equivalence for the code resulting from load sinking (Section 3.2).

Once these steps have been carried out, we are in a position to carry out the final step of unspeculation for the speculative region R:

1. Replace each speculative load in R by an unspeculative load.
2. Delete each speculation check in R.

Deleting the speculation check causes the corresponding control flow edge to the recovery code to be deleted as well. Usually, this causes the corresponding recovery code to become unreachable. Such unreachable code is detected and eliminated in the normal course of subsequent program analyses.

4 Experimental Results

We implemented our ideas within *ILTO*, a binary rewriting system we have created for manipulating and optimizing Itanium binaries [13]. We used a set of seven programs from the SPECint-2000 benchmark suite: *bzip2*, *gzip*, *mcf*, *parser*, *twolf*, *vortex*, and *vpr*, compiled using Intel's *ecc* compiler version 5.0.1, at optimization level `-O3` together with profile feedback. The resulting binaries contain a significant amount of control speculation.

The effectiveness of our unspeculation algorithm can be measured in two ways: quantitatively and qualitatively. First, there are situations—such as when the path independence condition is not met—where our algorithm will fail to unspeculate a region of code. Therefore we want to know how often our unspeculation algorithm succeeds in converting speculated code to non-speculated code. Second, since the goal of unspeculation is to make programs easier to understand, we need some way to gauge how successful our algorithm is in this respect.

To address the first question, we compare the proportion of speculative loads and speculation checks removed from each program by our algorithm. Table 1 shows the results of counting the number of (a) speculative loads and (b) speculation checks before and after speculation. It can be seen that our algorithm reduces the number of speculative loads and speculation checks by about 75% on average.

For the second question, we use the idea that a simpler control-flow graph is usually easier to analyze and understand than a more complicated one, and therefore one measure of how much our algorithm contributes to comprehension is the relative complexity of the CFG before and after unspeculation. To estimate complexity, we count the number of instructions, basic blocks, and edges between blocks in the program. The results of this experiment are shown in Table 2. This table shows that, on average, the number of instructions decreased by about 6%, the number of basic blocks decreased by about 13%, and the number of edges decreased by about 12% after unspeculation. For one benchmark, vortex, we saw a significantly larger decrease in the number of instructions, blocks, and edges — about 14.5%, 29.1%, and 25.6% respectively.

We are also interested in the effect that unspeculation has on performance. Since unspeculation attempts to undo

PROGRAM	BASIC BLOCKS			EDGES			INSTRUCTIONS		
	Orig. (B_0)	Unspec. (B_1)	Change (%) $(B_0 - B_1)/B_0$	Orig. (E_0)	Unspec. (E_1)	Change (%) $(E_0 - E_1)/E_0$	Orig. (I_0)	Unspec. (I_1)	Change (%) $(I_0 - I_1)/I_0$
bzip2	2509	2299	8.7	4188	3867	7.7	9259	8881	4.1
gzip	3189	2845	10.8	5297	4767	10.0	12957	12345	4.7
mcf	1118	956	14.5	1774	1533	13.6	4000	3715	7.1
parser	8866	7838	11.6	15891	14243	10.4	29779	27939	6.8
twolf	20543	17916	12.8	33083	29022	12.3	79469	74571	6.2
vortex	43641	30932	29.1	79658	59251	25.6	165189	141245	14.5
vpr	10570	9425	10.3	18805	16997	9.6	44319	42143	4.9
GEOM. MEAN:			12.9			11.9			6.3

Key: Orig: Original speculated code; Unspec: Unspeculated code

Table 2: Effects of unspeculation on program size

Program	*Execution Time* (sec)		T_1/T_0
	Original (T_0)	Unspeculated (T_1)	
bzip2	843.65	859.86	1.019
gzip	633.15	700.33	1.106
mcf	1409.94	1432.59	1.016
parser	1190.45	1268.94	1.066
twolf	1267.49	1333.95	1.052
vortex	835.32	839.26	1.005
vpr	906.85	985.82	1.087
GEOMETRIC MEAN			1.050

Table 3: Performance

Program	size (bytes)	*Processing time* (sec)		T_{uns}/T_{tot} (%)
		Unspeculation (T_{uns})	Total (T_{tot})	
bzip2	756848	2.565	99.730	2.57
gzip	783312	3.234	96.715	3.34
mcf	677712	2.765	87.653	3.15
parser	870032	5.350	125.541	4.26
twolf	1283968	90.856	277.396	32.75
vortex	2067440	79.756	430.865	18.51
vpr	1030080	8.468	146.303	5.79
GEOM. MEAN:				5.02

Table 4: Processing time

a compiler optimization, we expect that unspeculation results in less efficient code. To test this, we measured the execution times of the seven benchmarks before and after unspeculation. The programs were run on an HP i2000 workstation with a 733 MHz Intel Itanium processor running Redhat Linux 7.1 with 1 GB of main memory and 2 GB of swap space. Execution times for these programs were obtained as follows: Each binary was run five times on an unloaded machine and its runtime was measured using the Unix `time` command; the largest and smallest of the resulting run times were discarded; then the arithmetic mean of the remaining three execution times was computed and taken as the running time for that binary. We used statically linked binaries for our experiments, compiled with additional flags to instruct the linker to retain relocation information.[4] The results of these tests are shown in Table 3. This table shows that the unspeculated binaries suffer a performance hit of about 5% on average.

Figure 4 shows the time taken to carry out unspeculation. The primary goal of our current system is flexibility for experimentation, and processing speed is not a high priority. In particular, we use a simple file-based interface between the unspeculation module and the Omega calculator: the constraints generated are written out to a file, the Omega calculator is invoked on this file and the results written out to another file, which is then read back in by the unspeculator. The associated overheads lead to a significant increase in the cost of unspeculation: while they are around 2–4% of the total processing time for most of the benchmarks tested, they can be as high as 33%. We believe that a more efficient interface with the Omega calculator would reduce these costs significantly. On average, the cost of unspeculation is just over 5% of the total processing time.

[4]The requirement for statically linked executables is a result of the fact that *ILTO* relies on the presence of relocation information to distinguish addresses from data. The Unix linker `ld` refuses to retain relocation information for executables that are not statically linked.

5 Related Work

There is a large body of literature on reverse engineering, re-engineering, and program understanding (see, for example, [2, 3, 7, 10, 12]). Our work is complementary to, and supportive of, the traditional literature on reverse engineering and re-engineering: by undoing the effects of optimizations, it simplifies the task of reverse engineering highly optimized code containing speculative operations.

Also related is the work on debugging of optimized code (e.g., see [4, 8, 9]). This problem is similar to ours in the sense that when dealing with optimized code, a debugger must attempt to undo the effects of optimization and map the program state in the optimized code to source-level constructs in the original program. The technical issues that arise in this context, however, have to do with figuring out the relationship between the optimized executable and the original source code at runtime; by contrast, we examine static program transformation techniques for undoing optimizations. Moreover, our goals are fundamentally different, since they are aimed at reverse engineering rather than debugging. Finally, to the best of our knowledge none of these works address the problem of dealing with speculative execution.

6 Conclusions

While the speculative execution features of modern architectures such as the Intel Itanium can lead to significant performance improvements, they also lead to a considerable increase in the complexity of low level code. This can hinder reverse engineering and program comprehension of such codes. This paper describes a technique to transform speculative code into “normal” unspeculative code while preserving program semantics, thereby allowing more effective application of traditional reverse engineering and re-engineering techniques. Experiments indicate that our technique is effective: we are able to eliminate around 75% of the speculative loads and speculation checks in the programs tested.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., 1985.

[2] E. J. Byrne. A conceptual foundation for software re-engineering. In *International Conference on Software Maintenance*, pages 216–235, November 1992.

[3] E. J. Chikofsky and J. H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, pages 13–17, January 1990.

[4] Max Copperman. Debugging optimized code without being misled. *ACM Transactions on Programming Languages and Systems*, 16(3):387–427, May 1994.

[5] S. K. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *Proc. 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-98)*, pages 12–24, January 1998.

[6] S. S. Liao *et al.* Post-pass binary adaptation for software-based speculative precomputation. In *Proc. ACM SIGPLAN’02 Conference on Programming Language Design and Implementation (PLDI)*, June 2002.

[7] P. A. V. Hall. *Software Reuse, Reverse Engineering, and Re-engineering*, pages 3–31. Software Reuse and Reverse Engineering in Practice.

[8] J. Hennessy. Symbolic debugging of optimized programs. *ACM Transactions on Programming Languages and Systems*, 4(3):323–344, 1982.

[9] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proc. SIGPLAN ’92 Conference on Programming Language Design and Implementation (PLDI)*, pages 32–43, 1992.

[10] K. Lano and H. Haughton. *Reverse Engineering and Software Maintenance — A Practical Approach*. McGraw-Hill, 1994.

[11] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Comm. ACM*, 35:102–114, August 1992.

[12] M. Rekoff. On reverse engineering. *IEEE Transactions on Systems, Man and Cybernetics*, 3/4:244–252, 1985.

[13] N. Snavely, S. K. Debray, and G. R. Andrews. Predicate analysis and if-conversion in an Itanium link-time optimizer. In *Proc. Workshop on Explicitly Parallel Instruction Set (EPIC) Architectures and Compilation Techniques (EPIC-2)*, November 2002.