Unpredication, Unscheduling, Unspeculation: Reverse Engineering Itanium Executables*

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

Abstract

EPIC (Explicitly Parallel Instruction Computing) architectures, exemplified by the Intel Itanium, support a number of advanced architectural features, such as explicit instruction-level parallelism, instruction predication, and speculative loads from memory. However, compiler optimizations that take advantage of these features can profoundly restructure the program's code, making it potentially difficult to reconstruct the original program logic from an optimized Itanium executable. This paper describes techniques to undo some of the effects of such optimizations and thereby improve the quality of reverse engineering such executables.

Index Terms: reverse engineering, EPIC architectures, speculation, predication, code optimization

1. Introduction

There has been a great deal of recent interest in EPIC (Explicitly Parallel Instruction Computing) architectures, such as the Intel IA-64 (Itanium), which support advanced architectural features designed to get around low-level performance bottlenecks and enhance performance. Such features include explicit instruction-level parallelism, to increase throughput; predicated instructions, to reduce the need for explicit control transfers and associated pipeline stalls; and speculative loads, to reduce the latency associated with loads from memory. These features, moreover, are exposed to the compiler, which is responsible for generating code in a way that exploits them effectively.

In order to make effective use of the capabilities of EPIC architectures, compilers typically carry out a number of low-level optimizations. In particular, there are three optimizations that can significantly improve program performance: instruction scheduling, predication (or if-conversion), and speculation. *Instruction scheduling* refers to changing the order of instructions, where possible, in order to increase instruction-level parallelism. *Predication* refers to the selective elimination of conditional branches in favor of predicated instructions, thereby eliminating bubbles (resulting from control transfers) in the instruction pipeline. *Speculation* refers to the early execution of memory operations in order to hide their latency.

These optimizations can be very effective in exploiting the advanced architectural features of the underlying processor. However, they also profoundly restructure the low level code of programs, making it potentially difficult to reconstruct the original program logic from an optimized executable. This complicates the task of software systems that statically analyze or modify executable programs, e.g., reverse engineering systems, static binary translators, and link-time optimizers.

This paper is aimed specifically at the task of reverse engineering optimized Itanium binaries. There are many legitimate reasons why one might want to reverse engineer such files, the general reason being a need to understand

*A preliminary version of this paper appeared in the *Proceedings of the 10th. IEEE Working Conference on Reverse Engineering* (WCRE-2003) [21]. This work was supported in part by the National Science Foundation under grants CCR-0073394, EIA-0080123, and CCR-0113633.

the structure and/or working of software whose source code is not available. For example, a software vendor may wish to reverse engineer an existing product, made by a different vendor, in order to develop software interoperable with it. A vendor may reverse engineer software sold by another vendor to determine whether any of his patents are being violated. A user may wish to reverse engineer a software package to ensure that it does not contain any backdoors or other malware embedded within it. Note that in such applications, it is not important for the result of reverse engineering to be *identical* to the original source code (indeed, the very assumption that no source code is available implies that there is no way to determine whether it is): it simply has to be semantically equivalent to the original. Indeed, because of compiler transformations such as function inlining, common subexpression elimination, dead code elimination, etc., in general it is impossible to guarantee that a reverse engineering effort, starting with an executable, will recover the original source code (e.g., the result of function inlining, followed by constant propagation into the inlined code and then dead code elimination or common subexpression elimination, can bear very little resemblance to the original source code). A direct implication of this is that the techniques described in this paper do not promise to reconstruct the original source code, but rather simply a semantically equivalent version.

This paper presents algorithms for undoing many of the effects of scheduling, predication, and speculation, thereby simplifying the problem of identifying the original program structure and reasoning about its behavior. It is complementary to, and supportive of, traditional approaches to reverse engineering and re-engineering (e.g., see [2, 8, 14]): by undoing the effects of optimizations, it simplifies the task of reverse engineering highly optimized Itanium executables.

While there is a significant body of work on reverse engineering of binaries [1, 3, 4, 5, 6, 9, 13, 16, 18, 23], they typically deal either with interpreted bytecode formats or with executables on architectures, such as the Intel x86, that do not support features such as predication and speculation. We are not aware of any other work on reverse engineering that attempts to deal with such architectural features and the compiler optimizations that exploit them.

The remainder of the paper is organized as follows. Section 2 gives some background on the Itanium architecture and related compiler optimizations. Section 3 discusses a straightforward approach to reverse engineering Itanium executables. Section 4 describes a dataflow analysis for inferring relationships between predicate registers on the Itanium. Sections 5, 6, 7, and 8 discuss a number of low-level code transformations for undoing the effects of Itanium compiler optimizations. Section 9 provides experimental results, and Section 10 concludes.

2. Background

This section gives background information on the Intel Itanium architecture and on the ways compilers make use of predication, scheduling, and speculation. The reader familiar with the Itanium can skip this section and go directly to Section 3, which discusses a straightforward approach to reverse engineering Itanium executables.

The Itanium contains multiple functional units and uses programmer specified instruction-level parallelism. Every instruction is *predicated*: It specifies a one-bit predicate register, and if the value of that register is true (i.e., 1), then the instruction is executed; otherwise, it has no effect. For example, the instruction

```
(p6) add r15 = r15, r16
```

writes the sum of registers `r15` and `r16` into `r15` if the predicate register `p6` has the value 1 when this instruction is executed at runtime, and has no effect otherwise. The Itanium has 64 predicate registers; register `p0` is hard-wired to the constant value *true* (assignments to it are ignored). Many instructions in programs use `p0` as their predicate; these are said to be *unguarded* and by convention the predicate register is not specified in assembly code (as shown below). Instructions that specify a predicate register other than `p0` are said to be *guarded*. Conditional branches are also expressed using guarded branch instructions, e.g.:

```
        set p based on test
(p) br.cond TargetAddr
```

Predicate registers are set by compare instructions, which typically set them in complementary pairs: one register is set to indicate whether the condition being tested is true, the other is set to indicate whether it is false. There are three broad classes of compares: normal, unconditional, and parallel. A normal compare is of the form

(p) `cmp.`*rel* $dst_1, dst_2 = src_1, src_2$

where rel is a relation and dst_1 and dst_2 are predicate registers. It has the following semantics:

$$
\begin{array}{l}
\textbf{if}\ (\ p\ )\ \{ \\
\quad \textbf{if}\ (\ src_1\ rel\ src_2\ )\ \{ \\
\quad\quad dst_1 = 1;\ dst_2 = 0; \\
\quad \} \\
\quad \textbf{else}\ \{ \\
\quad\quad dst_1 = 0;\ dst_2 = 1; \\
\quad \} \\
\}
\end{array}
$$

For example, the instruction

```
(p6) cmp.eq p7,p8 = r10, r11
```

behaves as follows: if predicate register `p6` has the value 1, it sets the predicate registers `p7` and `p8` to 1 and 0, respectively, if `r10` and `r11` are equal, and to 0 and 1 if they are not. If `p6` is 0, the values of `p7` and `p8` are unaffected.

An unconditional compare has the form

$$(p)\ \texttt{cmp}\textbf{.unc.}rel\ dst_1, dst_2 = src_1,\ src_2$$

and has the following semantics:

$$
\begin{array}{l}
dst_1 = dst_2 = 0; \\
\textbf{if}\ (\ p\ )\ \{ \\
\quad \textbf{if}\ (\ src_1\ rel\ src_2\ )\ \{ \\
\quad\quad dst_1 = 1;\ dst_2 = 0; \\
\quad \} \\
\quad \textbf{else}\ \{ \\
\quad\quad dst_1 = 0;\ dst_2 = 1; \\
\quad \} \\
\}
\end{array}
$$

Thus, it is like a normal compare, except that it clears both predicate-register operands before doing the comparison.

A parallel-OR compare sets both predicate-register operands if the data comparison is true; otherwise neither predicate register is changed. A parallel-AND compare clears both predicate-register operands if the data comparison is false; otherwise neither predicate register is changed. Parallel compares are used to compute sequences of logical OR and logical AND operations.

Programs express instruction-level parallelism using *instruction groups*. Each group is a sequence of instructions that do not contain register dependencies and therefore can potentially be issued in parallel. Instructions are fetched in three-instruction "bundles" that are executed in parallel if possible. Predication, combined with careful instruction scheduling, can be used to eliminate explicit branch instructions and to increase the amount of instruction-level parallelism considerably.

Performance can be improved still further by using a feature called *speculation* to ameliorate the effects of long-latency memory operations [15]. The idea is to allow such instructions to be executed much earlier than would be possible in traditional architectures—possibly before it is even known whether the address that is being loaded from is a valid address. The hope is that initiating such expensive computations early will allow their results to be available by the time (if) they are needed. A speculative load is denoted by an opcode '`ld.s`', and has semantics similar to those of "ordinary" loads, except that exceptions (e.g., due to an invalid address or a page fault) are deferred for later handling. This is done by setting a special bit associated with the destination register of the load, called a `NaT` ("Not a Thing") bit if an exception occurs during speculative execution. The `NaT` bits are propagated by any instruction attempting to use the result of the load. Later, when the program reaches a point where the result of a speculative computation is needed, a special speculation check instruction (with opcode '`chk.s`') is used to determine whether the speculative computation succeeded: If the checked register has its `NaT` bit set, execution branches to recovery code generated by the compiler; otherwise, execution continues as normal. The recovery code re-executes the computation where speculation failed, then transfers control back to the regular program code.

While an aggressive optimizing compiler that takes full advantage of these architectural features can obtain significant performance improvements relative to traditional architectures, the resulting code can be quite obscure and difficult to understand and reverse engineer. The problem is that predication, aggressive instruction scheduling, and speculation can change the order and placement of instructions in a program dramatically. This results in code whose structure and operations bear little resemblance to that of the original source code. This is illustrated in Figure 1, which shows the code that might be generated, under different levels of optimization, for the following source code fragment, which iterates down a linked list computing a value:

```
while (ptr != NULL) {
  if (i == 0) {
    sum += ptr->data1;
  }
  else {
    sum -= ptr->data2;
  }
  ptr = ptr->next;
  i--;
}
```

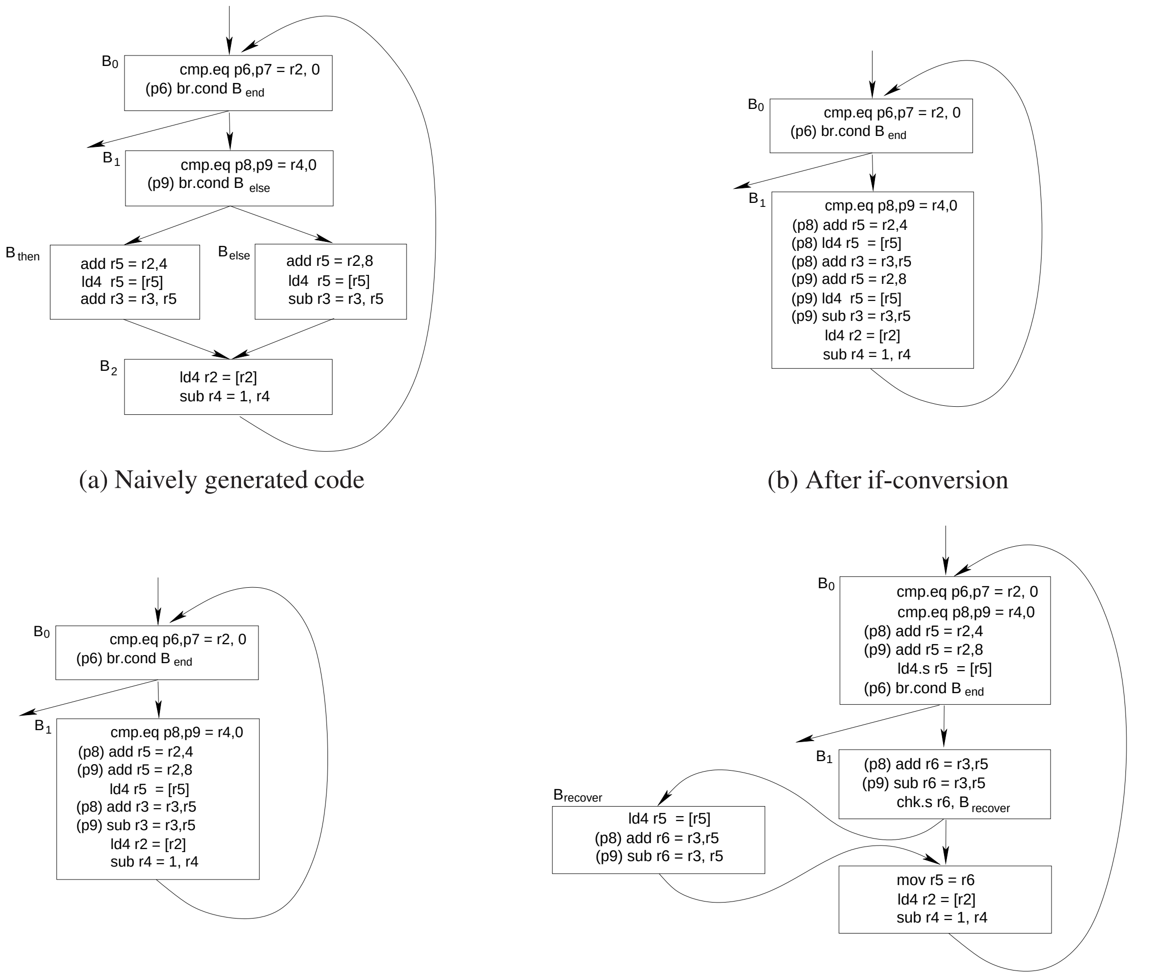
Figure 1(a) shows Itanium machine code generated in a straightforward way from this source code fragment. The logic of this code—involving the test and conditional branch to either of two distinct computations—is not difficult to figure out, and the code is correspondingly straightforward to reverse engineer using standard techniques.

Figure 1(b) shows the code resulting from applying if-conversion, i.e., predication, to the code from Figure 1(a). The conditional branch in block B1, and the conditionally executed instructions in blocks $\mathsf{B}_{\mathsf{then}}$ and $\mathsf{B}_{\mathsf{else}}$, have been replaced by a set of predicated instructions in B1. Note that this has eliminated a conditional branch (in block B_1) and an unconditional branch (in either B_{then} or B_{else}). Typically, branch instructions take several cycles to execute because the instructions at the branch target may not be immediately available in the CPU instruction pipeline. This results in a "bubble" in the instruction pipeline, i.e., one or more cycles when no useful instructions are executed. Replacing the branches with predicated instructions causes these bubbles to be eliminated. This can improve performance, but it obscures the logic of the computation because it requires a careful examination of the relationships between the values of the predicate registers `p8` and `p9` to determine what the computation is doing.

Figure 1(c) shows the result of applying instruction scheduling to the code from Figure 1(b). This code is better able to exploit instruction-level parallelism, but by rearranging the instructions so that instructions guarded by the same predicate register become separated, it makes the program harder to understand and reverse engineer.

Finally, Figure 1(d) shows the code resulting from applying speculation to the code from Figure 1(c). The resulting code is better able to hide the delays associated with memory load operations. However, it has two effects on code structure. First, the load operation that used to be in block B_1 is now moved across a conditional branch it depends on, into block B_0, where it may potentially fail (if register `r5` contains NULL). Second, additional code is added—the speculation check in block B_1, and the recover code in basic block $B_{recover}$ and associated control transfer—to recover from any such failures in the speculative code. These serve to further obscure the program logic.

Overall, it can be seen that the structure of the fully optimized code in Figure 1(d) is quite different from that of the original code in Figure 1(a). This makes it difficult to recover the original program logic from the code of Figure 1(d). This paper addresses low-level program analyses and code transformations that can be used to unravel the original structure of optimized Itanium code, thereby laying a foundation for the application of other, higher level, reverse engineering tools. Our goal is to take an optimized Itanium program and recover from it an "ordinary" control flow graph that is as simple as possible. This process consists of three transformations: *unpredication*, to undo the effects of predication and make control flow explicit (Section 5); *unscheduling*, to undo the effects of instruction scheduling and group related instructions together (Section 6); and *unspeculation*, to undo the effects of speculation and recover the original unspeculative code (Section 7). These transformations are assisted by *predicate analysis*, which infers relationships between predicate registers and allows us to improve the quality of our transformations (Section 4).



(a) Naively generated code

(b) After if-conversion

(c) After scheduling

(d) After speculation

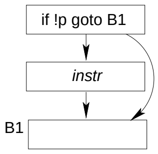
Figure 1. Itanium code under different optimizations

3. Naive Reverse Engineering of Itanium Executables

Recent years have seen the introduction of a number of architectures supporting *predication*, where the execution of an instruction can effectively be turned on or off dynamically using 1-bit predicate registers. Amongst the best known of these is the Itanium, though other architectures supporting predication include the Philips Trimedia, Texas Instruments TMS320C6x DSP, and the ARM. Predication has the effect of eliminating conditional branches, which can be an advantage architecturally, but which can also have the effect of obscuring the control flow logic of a program. For example, the structure of the predicated control flow graph shown in Figure 1(b) is significantly different from the control flow logic of the original program (Figure 1(a)). This affects program analyses and impedes program understanding. This suggests that the most fundamental component of any system that aims to reverse engineer an Itanium executable should be the removal of predication, i.e., the replacement of guarded instructions by a combination of unguarded instructions[1] and explicit control flow. This transformation is referred to as *unpredication* (sometimes called "reverse if-conversion").

One simple way to get rid of predication is to replace each predicated instruction by an unguarded instruction together with explicit control flow. Thus, each instruction of the form '`(p) instr`' is converted to code of the form

[1] Recall that, as discussed in Section 2, an unguarded instruction is one where the predicate register is hard-wired to the value *true*.



In carrying out this transformation, a contiguous sequence of instructions where each instruction is guarded by the same predicate register p, and none of which modify p, can all be put into the same basic block, preceded by a single conditional branch on p.

This straightforward transformation eliminates guarded instructions, but it has the effect of introducing a great many new basic blocks and control flow edges. This results in a large and messy control flow graph, with a great many unfeasible paths that serve to obscure program logic and adversely affect program analyses. The next two sections discuss how this problem can be mitigated.

4. Predicate Analysis

This section sketches a dataflow analysis we use to derive relationships between predicate registers. This information is then used to improve the quality of reverse engineering, as discussed in the remainder of the paper.

Our analysis reasons about two kinds of relationships between predicate registers. Let P and Q be two predicate registers, $\Rightarrow$ denote logical implication and $\Leftrightarrow$ denote logical equivalence, i.e., $x \Leftrightarrow y$ iff $x \Rightarrow y$ and $y \Rightarrow x$. We have the following definitions:

Complementarity: P and Q are *complementary* at a program point if $P \Leftrightarrow \neg Q$, i.e., exactly one of them must be true when control reaches that point.

Dominance: P *dominates* Q if $Q \Rightarrow P$. This means that if Q is false then P must be false as well, i.e., Q can only be true if P is also true.

Our implementation also keeps track of the weaker property of *disjointness*: two predicates P and Q are disjoint at a program point if $\neg(P \wedge Q)$ whenever control reaches that point, i.e., they cannot both be true at the same time. While disjointness information is useful, e.g., for instruction scheduling [20], it is not central to this discussion and therefore is not pursued further here.

An an example, the following instruction sets predicate registers `p6` and `p7` to complementary values, depending on whether `r5` $\leq$ `r6`:

```
cmp.le p6,p7=r5,r6
```

Immediately after this instruction, `p6` and `p7` are complementary, regardless of their actual values. Suppose that the next instruction that alters `p6` or `p7` is

```
(p8) cmp.eq p6,p7=r10,r11
```

This instruction is executed conditionally, depending on whether `p8` is true. However, `p6` and `p7` will still be complementary, even though their values might have changed.

Dominance relations typically arise from unconditional compare instructions: e.g., predicate registers `p6` and `p7` are both dominated by `p8` after the instruction

```
(p8) cmp.unc.eq p6,p7=r10,r11
```

Conceptually, dominance between predicate registers corresponds to nested conditionals in terms of control structure.

Example 4.1 The C code:

```
if (x == 0) {
  if (y == 0)
    z = 1;
  else
```

```
    z = 2;
  }
  else {
    z = 3;
  }
```

corresponds to the following Itanium code:

```
     cmp.eq P,Q = x,0            # I1
(P)  cmp.unc.eq R,S = y,0        # I2
(R)  mov z = 1                   # I3
(S)  mov z = 2                   # I4
(Q)  mov z = 3                   # I5
```

After instruction `I1`, `P` is 1 and `Q` is 0 if `x` is 0, while otherwise `P` is 0 and `Q` is 1. Recall that an unconditional compare instruction first clears its destination registers, after which, if its predicate register ha the value 1, it sets its destination registers appropriately based on the condition and the operand values (see Section 2). It follows that after instruction `I2` we have the following relationships between the registers:

$$\mathtt{R} = \begin{cases} 1 & \text{if } \mathtt{P} = 1 \text{ and } \mathtt{y} = 0 \\ 0 & \text{if } \mathtt{P} = 1 \text{ and } \mathtt{y} \neq 0 \\ 0 & \text{if } \mathtt{P} = 0 \end{cases}$$

$$\mathtt{S} = \begin{cases} 0 & \text{if } \mathtt{P} = 1 \text{ and } \mathtt{y} = 0 \\ 1 & \text{if } \mathtt{P} = 1 \text{ and } \mathtt{y} \neq 0 \\ 0 & \text{if } \mathtt{P} = 0 \end{cases}$$

It can be seen that each of the predicate registers `R` and `S` is true only if `P` is true, whence it follows that both `R` and `S` are dominated by `P`. Note that this reflects the nesting structure of original control flow in the source code, where the predicate '`y == 0`' is evaluated only if '`x == 0`' is true. ■

Our predicate analysis is a forward dataflow analysis that propagates sets of pairs of predicates (p, q) over the control flow graph of a function. For simplicity of exposition, we discuss only the propagation of complementarity relations here; the propagation of dominance relations is conceptually similar, and discussed in more detail elsewhere [20]. The set $\mathsf{IN}(B)$ denotes the set of pairs of complementary predicates at the entry to block B, and $\mathsf{OUT}(B)$ the set of pairs of complementary predicates at the exit from B.

Let B_0 denote the entry block of the function under consideration. The following dataflow equations specify how the above four sets are computed.

1. Determining complementarity relationships at the entry to a block B involves three cases:

   (a) For intraprocedural analysis we assume that nothing is known at the entry block B_0 to a function:

   $\mathsf{IN}(B_0) = \emptyset$.

   (b) If B is the return block for a call to a function f from a block B', then the dataflow information entering B is obtained by taking the complementarity relations that hold at exit from B', i.e., just before control is transferred to f, and filtering this through the summary information known about the behavior of the callee function f:

   $\mathsf{IN}(B) = \mathsf{FnOut}_f(\mathsf{OUT}(B'))$.

   (c) Otherwise, it consists of the complementarity relations that hold at the exit from each of B's predecessors, and so are guaranteed to hold at entry to B:

   $$\mathsf{IN}(B) = \bigcap_{P \in preds(B)} \mathsf{OUT}(P).$$

2. The dataflow information at the exit from a basic block B is obtained by taking the dataflow information $\mathsf{IN}(B)$ entering B and propagating it through B to compute $\mathsf{OUT}(B)$ as a function of $\mathsf{IN}(B)$ and the instructions in B. The details of this computation are given in Figure 2.

$$\mathsf{IN}(B) \; = \begin{cases} \emptyset & \text{if } B = B_0 \\ \bigcap_{P \in preds(B)} \mathsf{OUT}(P) & \text{otherwise} \end{cases}$$

$S = \mathsf{IN}(B)$;
for each instruction I in basic block B in their order of occurrence in B **do**
  if I is not a compare instruction **then continue**;

  let I be of the form '*(pG)* cmpOp *pA,pB = operands*';
  $isComp = (pA, pB) \in S$;   */* $isComp$ indicates whether S contains (pA, pB) at this point. */*
  remove all pairs containing pA or pB from S;
  if I is a normal compare instruction **then**
    if *isComp* **or** $pG \equiv$ p0 */* I is unguarded */* **then**
      add (pA, pB) to S;
    end if
  end if
  */**
  ** At this point, S contains (pA, pB) iff (i) I is a normal compare, and (ii) either $(pA, pB) \in S$ at the point*
  ** immediately before I, or I is unguarded. Note that this implies, in particular, that if I is an unconditional*
  ** or parallel compare instruction, then all complementarity information involving the destination registers*
  ** pA or pB are deleted from S*
  **/*
end for

$\mathsf{OUT}(B) \; = S$;

Figure 2. Computing Predicate Complementarity Sets for a Basic Block B

We solve the dataflow equations given above by starting with the initial values $\mathsf{IN}(B) \; = \mathsf{OUT}(B) \; = \emptyset$ for all basic blocks B in the function under consideration, and then computing a fixpoint by iteratively applying the equations above until there is no change to any of these sets.

In case 1(b) of the dataflow equations above, $\mathsf{FnOut}_f(S)$ denotes the effect of the function call f on the complementarity relations at the call site. A simple conservative estimate for intra-procedural analyses is to assume that nothing is known about complementarity relationships at the return from a function call, i.e., $\mathsf{FnOut}_f(S) = \emptyset$ for all f and S. We can do better, however, by identifying, for each function f whose complete call graph is available for analysis, the set $\mathsf{Unchg}(f)$ of predicate registers whose values will not be affected by a call to f. This is done as follows:

1. Define $\mathsf{SaveRestore}(f)$ to be the set of predicate registers that are saved at entry to f before any use, and restored prior to leaving f. These sets can be determined by inspecting the prolog and epilog of f's code.

2. Let $\mathsf{Unchg}(B)$ be the set of predicate registers whose values will not be changed during the execution of B:

$$\mathsf{Unchg}(B) \; = \begin{cases} \emptyset & \text{if } B \text{ ends in a function call} \\ \{p \mid p \text{ not assigned to in } B\} & \text{otherwise} \end{cases}$$

Then, the set of predicate registers that are unaffected by a call to f is given by

$$\mathsf{Unchg}(f) \; = \mathsf{SaveRestore}(f) \bigcup \big(\bigcap_{B \in blocks(f)} \mathsf{Unchg}(B) \big).$$

Note that the set $\mathsf{Unchg}(f)$ can be computed in a single pass over the instructions of f. We can then define the effect of a call to a function f on predicate complementarity relationships as follows:

$$\mathsf{FnOut}_f(S) = \{(p, q) \in S \;\mid\; \{p, q\} \subseteq \mathsf{Unchg}(f)\ \}.$$

This is a pessimistic estimate of the effects of a function call, because when computing $\mathsf{Unchg}(B)$ for a basic block B, we assume that all predicate registers may be overwritten if B contains a function call. A better approach, which we have implemented, is to propagate $\mathsf{Unchg}(f)$ values over the call graph of the program and iterate to a fixpoint.

To reason about the soundness of this dataflow analysis, we first observe that the algorithm shown in Figure 2 for analyzing a single basic block is sound. In other words, if $(pA, pB) \in \mathsf{OUT}(B)$ for some basic block B, then either $(pA, pB) \in \mathsf{IN}(B)$ and B does not contain any compare instructions that assign to either pA or pB; or B contains a compare instruction that assigns complementary values to pA and pB that can reach the end of B. The soundness of the overall iterative algorithm then follows directly from the fact that the analysis is a monotone (in fact, distributive) dataflow analysis [11, 12]. Function calls are handled by a simple examination of registers that are not changed by the call, and so do not affect soundness. Since the IN and OUT sets consist of pairs drawn from a finite set of predicate registers, they are finite as well, whence termination follows from the monotonicity of the operators used in the analysis.

Unlike other proposals for inferring relationships between predicates in Itanium-like EPIC processors [7, 10, 19], our analysis is formulated within the framework of a traditional meet-over-all-paths dataflow analysis. Thus, it is relatively straightforward to understand, implement, and extend in various ways, e.g., to inter-procedural analysis. Our current implementation extends the algorithm described above to a simple context-insensitive inter-procedural analysis.

5. Intelligent Unpredication

The naive unpredication algorithm described in Section 3 has the disadvantage of creating a large and messy control flow graph. Here we describe a more intelligent approach to unpredication that utilizes the results of the predicate analysis described in Section 4. We consider "predicate groups," which are instructions whose predicate registers are related based on information obtained from predicate analysis. More formally, a *predicate group* is defined to be a maximal sequence of consecutive predicated instructions $(p_1)I_1, (p_2)I_2, \ldots, (p_n)I_n$ within the same basic block, such that for any pair of predicate registers p_i, p_j guarding instructions in the sequence, one of the following holds: (i) $p_i = p_j$; (ii) some p_k dominates both p_i and p_j; or (iii) p_i and p_j are complementary. As a special case, a sequence of unguarded instructions forms a predicate group, since their predicates (all `p0`) are identical.

Given a set of dominance relationships $\mathcal{D}$ inferred at a program point via predicate analysis, let $(p_1, p_2, \ldots, p_k)$ be a maximal sequence of predicates such that

$$\mathcal{D} \models p_1 \Rightarrow p_2 \Rightarrow \cdots \Rightarrow p_k$$

where $\models$ denotes logical entailment. In other words, from $\mathcal{D}$ we have p_k dominates p_{k-1}, ..., p_2 dominates p_1. We refer to such a maximal chain of dominance relations as a *dominance chain* for the predicate p_1; the last element p_k of the chain is referred to as its *anchor*. Recall that, as discussed in the previous section, dominance relations between predicate registers in a program reflect nested conditionals in the control flow of the original program. Thus, given an instruction $I \equiv$ '(p) `instr`,' the dominance chain for p makes explicit the control flow nesting corresponding to the predicates that affect the execution of instruction I. Suppose that this dominance chain is $(p, p_1, \ldots, p_n)$; this means—from the definition of the dominance relation—that I will be executed only if each of the predicates $p, p_1, \ldots, p_n$ is true. We can state this explicitly by writing the instruction I as[2]

$(p, p_1, \ldots, p_n)$ `instr`.

Once dominance chains have been made explicit, we can use them to identify predicate groups based on complementarity relationships between the anchors of these dominance chains, and carry out unpredication on these predicate groups. The algorithm for this is shown in Figure 3.

To reason about the soundness of this algorithm, suppose that G_B is the unpredicated control flow graph obtained by applying the algorithm to a predicated basic block B. A simple induction on the depth of recursion of the algorithm of Figure 3 suffices to show that G_B is equivalent to B: the base case is trivial, and the main issue in the inductive

[2]Note that this is simply an internal representation, within a software tool, of an instruction, intended to make some aspects of its runtime behavior explicit for program transformation purposes. The instructions actually executed on the Itanium hardware still have just a single predicate register.

Input: A basic block B where each instruction has its dominance chain made explicit.

Output: A control flow graph G_B corresponding to an unpredicated version of B.

Algorithm:

initially G_B consists of the single basic block B.

while there is any change to the control flow graph G_B being constructed **do**:

1. Find the longest sequence of contiguous instructions $S \equiv I_1, \ldots, I_n$ in B such that the anchor for the dominance chain of each instruction I_i in S is either p or q, for distinct predicate registers p and q that are complementary to each other.

   If there is no nonempty sequence of such instructions, return.

2. Partition the instruction sequence S into two sequences S_p and S_q, where S_p consists of those instructions in S whose dominance chain has anchor p, and S_q consists of those with anchor q. (Note that either of S_p and S_q may be empty.)

   This may require some rearrangement of instructions. However, this is permissible as long as the relative order of the instructions in S_p and S_q is consistent with their original ordering within S. The reason is that since p and q are complementary, when the instruction sequence S is executed, either those in S_p, or those in S_q, will be executed, but not both.

3. Let $\widehat{S}_p$ and $\widehat{S}_q$ be instruction sequences derived from S_p and S_q, respectively, as follows. Each instruction of the form

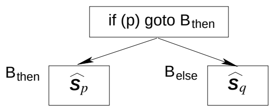
   $(p_1, \ldots, p_k, p)\ I$

   in S_p corresponds, in $\widehat{S}_p$, to an instruction of the form

   $(p_1, \ldots, p_k)\ I.$

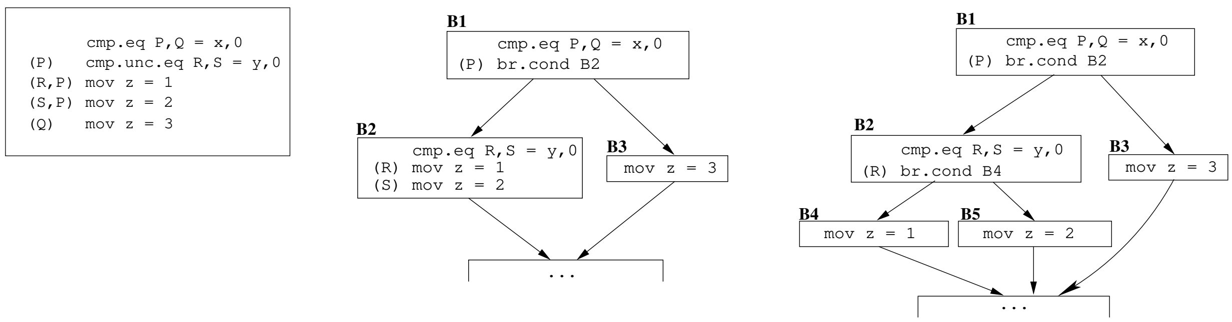
   $\widehat{S}_q$ is defined analogously. In other words, $\widehat{S}_p$ is obtained by deleting the anchor predicate from the dominance chain of each instruction in S_p; and similarly for $\widehat{S}_q$.

4. Replace the instruction sequence S by the following control flow structure, adjusting block B appropriately:



5. Recursively unpredicate each resulting basic block.

Figure 3. An algorithm for intelligent unpredication



(a) Initial basic block (b) after first iteration (c) after second iteration

Figure 4. An example of intelligent unpredication

step is showing that the transformation resulting from a single level of recursion preserves semantic equivalence. To this end, consider a maximal sequence of instructions $S \equiv I_1, \ldots, I_n$, found in step 1 of the algorithm, such that the anchors p and q of the dominance chains of these instructions are complementary. Let S_p consist of those instructions in S that have p as the anchor of their dominance chains, and S_q consist of those whose anchor is q. Since p and q are complementary, it follows that

S_p will be executed iff p is true at the beginning of S
iff q is false at the beginning of S
iff S_q is not executed.

This means that the instruction sequence S is semantically equivalent to

if (p) **then** S_p **else** S_q. (*)

Furthermore, considering the conditional (*), if S_p is executed it must be the case that p is true, which means that the anchor predicate register p can safely be deleted from the dominance chains of instructions within S_p. Reasoning similarly, the anchor predicate register q can be deleted from instructions within S_q. Note that this results in the code fragment

if (p) **then** $\widehat{S}_p$ **else** $\widehat{S}_q$

resulting from a single level of recursion in the algorithm of Figure 3, which establishes that a single level of recursion is semantics-preserving. It follows, by induction on the depth of recursion, that the unpredication algorithm preserves semantic equivalence. Termination follows from the fact that at each level of recursion, there is a decrease in either the length of the instruction sequence under consideration, or the length of their dominance chains, or both.

Example 5.1 Making dominance chains explicit on the instruction sequence shown in Example 4.1 yields:

```
      cmp.eq P,Q = x,0
  (P) cmp.unc.eq R,S = y,0
(R,P) mov z = 1
(S,P) mov z = 2
  (Q) mov z = 3
```

The effect of applying the algorithm of Figure 3 to this basic block is shown in Figure 4.

In the first iteration, there is a single predicate group in the block consisting of the three instructions predicated on `P` and the one predicated on `P`'s complement, `Q`. The first iteration splits these four instructions into two blocks as shown in figure 4(b). The three instructions that had been predicated on `P` have been put in the *then*-block, and `P` has been removed from their dominance chains. The instruction that had been predicated on `Q` has been moved into the *else*-block, and `Q` has been removed from its dominance chain.

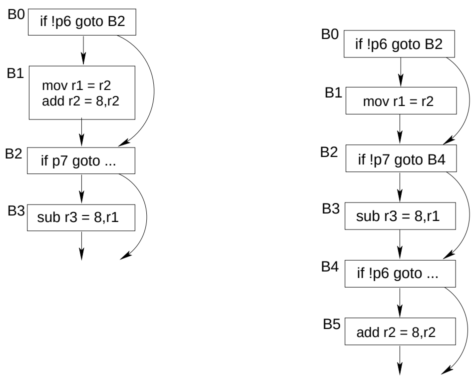
The unpredication algorithm then recursively processes the basic blocks so obtained. The resulting control flow graph is shown in figure 4(c). At this point, only conditional branches are predicated, so the algorithm terminates. Note that the final result here is isomorphic to the control flow structure of the source code shown in Example 4.1. ■

6. Unscheduling

Aggressive instruction scheduling permutes instructions within basic blocks. This can result in code whose behavior is difficult to understand and reason about. As an example, consider the code:

```
(p6) mov r1 = r2
(p6) add r2 = 8,r2
(p7) sub r3 = 8,r1
```

Even if we know nothing about the relationship between predicates `p6` and `p7`, it is easy to see that the first two instructions are both controlled by `p6`, and that the code therefore has the control flow structure shown in Figure 5(a). However, suppose that instruction scheduling permutes this instruction sequence to the following:



(a) without scheduling (b) with scheduling

Figure 5. Effect of instruction scheduling on reverse engineered control flow graphs

```
(p6) mov r1 = r2
(p7) sub r3 = 8,r1
(p6) add r2 = 8,r2
```

The control flow structure we would infer for this code sequence, shown in Figure 5(b), is much more complex. In addition, the resulting control flow graph has unfeasible paths (e.g., the path $B_0 \rightarrow B_2 \rightarrow B_4 \rightarrow B_5$), which can have an adverse effect on program analyses and hamper program understanding.

The goal of unscheduling is to group together related instructions that may have been separated during scheduling (it is possible for this to also group together code fragments that had been separate in the original program). Put another way, the unscheduler seeks to permute instructions within a basic block so as to minimize the number of predicate groups in that block (see Section 5 for a definition of predicate groups). It does so by *merging* predicate groups whenever possible. Two predicate groups A and B can be merged if (i) each of the predicates that appear in A is related to each of the predicates that appear in B and (ii) it is possible to move A and B so that they are adjacent. A predicate group can be moved past an adjacent group as long as no dependencies exist between their instructions. Assuming all other groups remain in place, a predicate group can occupy a range of positions whose boundaries are either dependent predicate groups or the boundaries of the basic block containing that group.

Our algorithm has two stages. First it finds the forward range of each predicate group and looks for another group in that range with which the first can be merged. Then it does the same for the backward range. The forward range scan algorithm is described in more detail in Figure 6; the backward range scan is completely analogous. The scan must be done in both directions because the process of merging two groups can be asymmetric; that is, it is possible that a group G cannot be moved forward to another group G', but that G' can be moved backward to meet G, or vice-versa. Recall the fragment

```
(p6) mov r1 = r2     /* group G1 */
(p7) sub r3 = 8,r1   /* group G2 */
(p6) add r2 = 8,r2   /* group G3 */
```

The first predicate group, G1, cannot be moved down to meet and merge with group G3, since there is a dependent instruction in the way. However, G3 can be moved back to meet and merge with G1.

To see that this transformation preserves program semantics, note that two non-adjacent predicate groups G and G' are merged only if one of them can be moved past the intervening code, thereby making G and G' adjacent instruction sequences, without violating any dependencies between instructions. Suppose that G' is moved past the intervening instruction sequence G'', i.e., there are no dependencies between any instruction in G' and any instruction in G''. Since both G' and G'' are within the same basic block B, they will both be executed if B is executed. This, together with the fact that there are no dependencies between G' and G'', mean that the relative order of execution between G' and G'' does not affect the behavior of the program. A similar argument applies if G is moved past G''. It follows that the transformation is semantics-preserving.

Input: A basic block B.

Output: The block B with its instructions rearranged so as to minimize the number of predicate groups, while leaving its behavior unchanged.

Algorithm:

1. [*Initialization*] Place each instruction in B in a separate predicate group of its own.
2. [*Forward pass*]

   while there is no change to B **do**
     for each predicate group G in B in reverse order **do**
       for each predicate group G' following G **do**
         if some instruction in G' is dependent on an instruction between G and G' **then**
           break
         elsif the predicate registers of G and G' are related **then**
           move G' immediately after G and merge them into a single predicate group;
           break
         endif
       endfor
     endfor
   endwhile

3. [*Backward pass*]
   (analogous to forward pass: we traverse the predicate groups G in forward order, and consider mergability with predicate groups G' preceding G.)

Figure 6. The Basic Unscheduling Algorithm

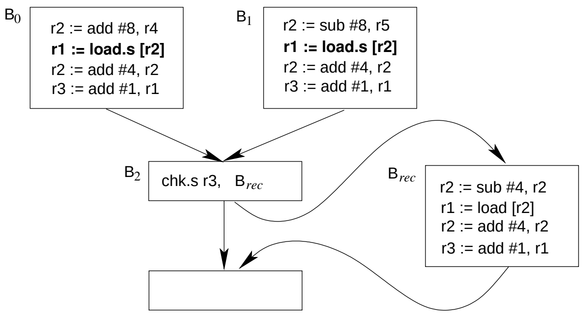
An important point to note here is that the notion of "dependence" between instructions, which plays a central role in the unscheduling algorithm, should take predication into account. The usual notion of dependence is that two instructions I and J are dependent if either one can write to a (register or memory) location that may be read from or written to by the other. When applied to predicated instructions, we can use the results of predicate analysis, described in Section 4, to refine this notion, as follows: Two instructions I and J, guarded by predicate registers p and q respectively, are dependent if (i) predicate registers p and q are not known to be disjoint; and (ii) either of them may write to a location that may be read from or written to by the other.

7. Unspeculation

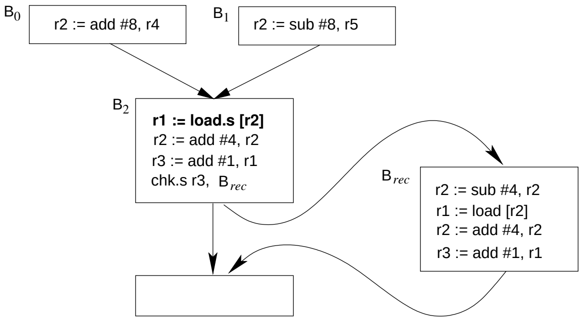
As discussed in Section 2, the main difference between speculative and unspeculative loads is that any exceptions raised by the former are deferred via the `NaT` bits. Our approach to unspeculation consists of two distinct phases. First, we move each speculative load to one or more points in the code stream where it can potentially be replaced by an unspeculative load operation. We call this *load sinking*. The details of load sinking are discussed in Section 7.1; for technical reasons, as discussed below, this has to be done together for groups of "related" speculative loads and speculation checks called speculative regions. 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. This is discussed in Section 7.2. Each of these steps must, of course, be semantics-preserving.

Once these steps have been carried out, we replace each speculative load in the speculative region by an unspeculative load, and delete each speculation check in that region. 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 analysis and optimization.

A more comprehensive discussion of this transformation appears elswehere [22].



(a) Before load sinking



(b) After load sinking

Figure 7. An example of load sinking

7.1 Load Sinking

The appearance of a speculative load in a program indicates that it cannot be guaranteed to execute without any exceptions. Thus, simply replacing a speculative load by an unspeculative one may not 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 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. The process of moving speculative loads "down" to their check instructions is referred to as *load sinking*, and is illustrated in Figure 7.

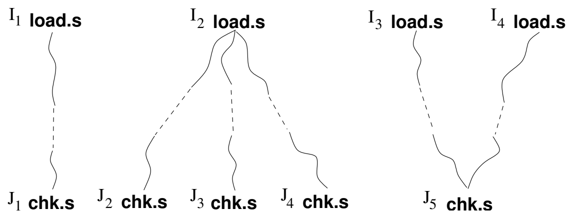


Figure 8. General structure of speculative computations

There are two aspects of speculative code that complicate load sinking. First, it is possible to do various operations, e.g., arithmetic, on the results of a speculative load: if the speculation fails, the resulting `NaT` bit is propagated by such operations. The second is that speculative loads and speculation checks need not even be in one-to-one correspondence: a particular speculative load may have several associated checks, and a speculation check may correspond to several different speculative loads (see Figure 8). The first of these means that when carrying out load sinking, it may be necessary to move not just the speculative load instruction, but other instructions that depend on it, as shown in Figure 7. The second aspect means that if a speculation check is associated with multiple speculative loads, we have to make sure that the set of instructions that has to be sunk to that check is the same for each of the associated speculative loads. We do this by grouping speculative loads and speculated checks into *speculative regions* that satisfy the following property:

(i) if x is a speculative load in a speculative region R and y is a speculation check on x, then y is in R; and

(ii) if y is a speculation check in R and x is a speculative load that is checked by y, then x is in R.

Intuitively, speculative regions capture the closure of checker and checkee relationships between speculative loads and associated speculation checks.

For each speculative region R, we check that for any speculation check C in R, the set of instructions I that must be sunk to C is the same regardless of which associated speculative load in R we consider. We refer to this condition as *path independence*. If this condition is satisfied, load sinking is effected by deleting the instruction sequence I from each speculative load in R and inserting at the beginning of each speculation check in R.

The code structure resulting from load sinking is illustrated in Figure 9.

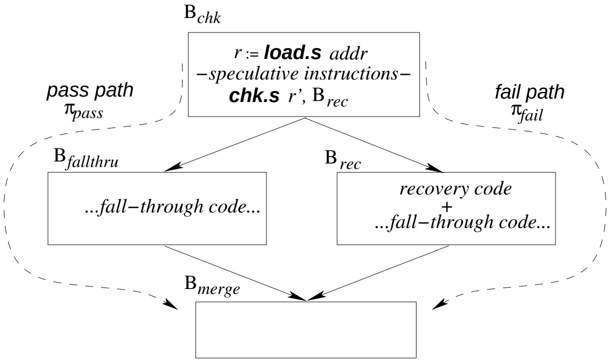


Figure 9. Code structure after load sinking

7.2 Recovery Code Verification

In Figure 9 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* $\pi_{fail} \equiv B_{chk} \rightarrow B_{rec} \rightarrow B_{merge}$.

In general, 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 B_2 in Figure 7(b). To recover if the load fails, the correct address has to be recomputed before reissuing the load, and so the recovery code needs extra instructions to fix the program state appropriately. The first instruction in the recovery code (block B_{rec}) 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 (in this case, register r_3) that were computed using the result of the speculative load, and also resets the value of registers (in this case r_2) whose values had to be changed to reissue the 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. 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.

Proving path equivalence involves reasoning about the contents of registers and memory locations along the pass and fail paths. While doing this, our implementation currently handles only case where each of the pass path π_{pass} and the fail path π_{fail} is a single straight-line path with no branches: if either π_{pass} or π_{fail} contains branches, the analysis conservatively assumes that path equivalence does not hold. 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. Our implementation is also 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. As the experimental results reported in Section 9 indicate, these assumptions suffice for most instances of speculation encountered in practice.

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.

There are three components to the formula Φ: Ψ_p, which expresses the values of locations at the end of the pass path; Ψ_f, which expresses the values of locations at the end of the fail path; and Δ, which combines values from Ψ_p and Ψ_f to state 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. We first define how these formulae are constructed, then describe how they are composed to give the formula Φ.

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 $C_k \;\equiv\; r_k^p = \mathit{mem}(s_j^p)$ 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:

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

Let $S = \{r_0 \mid r \text{ is a register used along the pass or fail path prior to being defined}\}$, i.e., S denotes the initial values of registers that are used along either the pass o the fail path. Then, the formula Φ is defined as

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

This quantification asserts that there exist some incoming register values for which path equivalence may not hold. If we can then show that Φ is unsatisfiable, it follows that path equivalence must hold for all possible values of the registers. Note that this is conservative: for example, it may be that a particular register always has the value 7 when control enters the code segment of interest, or is always divisible by 4, but the constraint above does not take such invariants into consideration. We could extend our ideas to take such invariants into account by adding conjunctively to the formula above; this is somewhat orthogonal to the central focus of our discussion, however, and we do not pursue it further.

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 [17] to determine the satisfiability of the formula Φ.

Applied to the recovery code shown in Figure 7, we get $\Phi = (\exists S)[\Psi_p \wedge \Psi_f \wedge \Delta]$, where $S = \{r2_0\}$, and:

$$
\begin{aligned}
\Psi_p = \quad & r1_1^p = mem(r2_0) \wedge r1_e^p = r1_1^p \\
& \wedge r2_2^p = r2_0 + 4 \wedge r2_e^p = r2_2^p \\
& \wedge r3_3^p = r1_1^p + 1 \wedge r3_e^p = r3_3^p.
\end{aligned}
$$

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

$$
\Delta = \; r1_e^p \neq r1_e^f \vee r2_e^p \neq r2_e^f \vee r3_e^p \neq r3_e^f
$$

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

$$
\begin{aligned}
\Psi_p \quad & \equiv \quad r1_e^p = mem(r2_0) \wedge r2_e^p = r2_0 + 4 \wedge r3_e^p = mem(r2_0) + 1 \\
\Psi_f \quad & \equiv \quad r1_e^f = mem(r2_0) \wedge r2_e^f = r2_0 + 4 \wedge r3_e^f = mem(r2_0) + 1.
\end{aligned}
$$

The overall constraint Φ, after simplification, is therefore the following:

$$
\begin{aligned}
(\exists r2_0)[\quad & r1_e^p = mem(r2_0) \wedge r2_e^p = r2_0 + 4 \wedge r3_e^p = mem(r2_0) + 1 && \text{/* } \Psi_p \text{ */} \\
& \bigwedge r1_e^f = mem(r2_0) \wedge r2_e^f = r2_0 + 4 \wedge r3_e^f = mem(r2_0) + 1 && \text{/* } \Psi_p \text{ */} \\
& \bigwedge (r1_e^p \neq r1_e^f \vee r2_e^p \neq r2_e^f \vee r3_e^p \neq r3_e^f)] && \text{/* } \Delta \text{ */}
\end{aligned}
$$

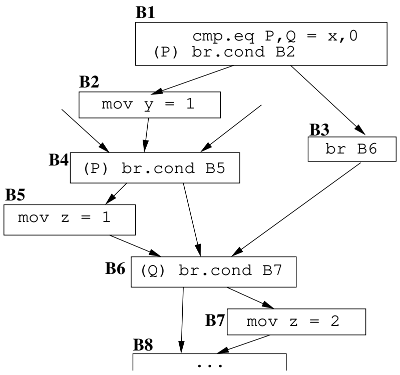
It is not difficult to see, from this, that the Δ constraints are not satisfiable, which implies that Φ is also unsatisfiable. This, in turn, implies path equivalence for the code in Figure 7.

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.

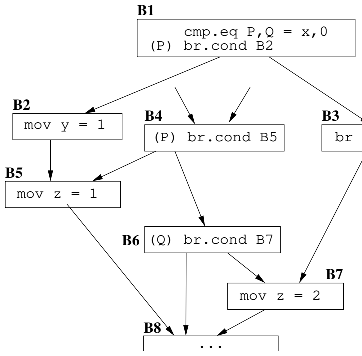
7.3 The Unspeculation Transformation

The overall unspeculation transformation consists of the following sequence of steps:

1. Group the speculative loads and speculation checks into speculative regions.
2. For each speculative region R:
   (a) Verify path independence for R. If path independence cannot be verified, abandon unspeculation for R.
   (b) Carry out load sinking.
   (c) Verify path equivalence and load equivalence for the code resulting from load sinking.
   (d) Replace each speculative load in R by an unspeculative load, and delete each speculation check in R.



(a) Unpredicated CFG before path simplification



(b) Unpredicated CFG after path simplification

Figure 10. An example of path simplification

Deleting the speculation check in the final step of the algorithm 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.

The correctness of the overall unspeculation transformation follows from the fact that each step of the transformation preserves program semantics. The correctness of load sinking follows from path independence, which implies that the instruction sequence I consisting of a speculative load, and comprising all the instructions that depend on that load, is the same for every speculative load and speculation check in a speculative region. This implies that the instructions that are executed on any path between a speculative load and a speculation check in that region are the same before and after sinking. Furthermore, since all instructions (transitively) dependent on the specilative load are sunk, it follows that the transformation does not affect the order of dependent instructions, i.e., does not violate any dependencies between instructions. It follows from this that load sinking does not affect program semantics. After load sinking, the path equivalence and load equivalence conditions ensure that the behavior of the program is the same whether the recovery code is executed or not, which implies that it suffices to retain just one of these two execution paths. Our transformation retains the pass path and eliminates the fail path.

8. Path Simplification

The discussion on intelligent unpredication in Section 5 focused on exploiting predicate relationships within a basic block. It turns out that the control flow graphs obtained from this algorithm can be further simplified using knowledge of predicate relationships across basic block boundaries. This is illustrated by the following code:

```
Begin:
     cmp.eq P,Q = x,0
 (P) mov y = 1
 (Q) br.cond After
Fallthrough:
 (P) mov z = 1
After:
 (Q) mov z = 2
```

This fragment consists of three blocks, each of which will be unpredicated separately. Unpredication produces the control-flow graph shown in Figure 10(a).

In this control-flow graph there are multiple paths that can never be taken. For instance, it is impossible that blocks B_2, B_4, B_6 are executed in that order, because B_2 can only be reached if P is true – hence, if B_4 is reached from B_2, the branch in B_4 must always be taken. So, nothing prevents us from redirecting the edge from B_2 to B_4 to instead point to B_5. We call such a redirection a *path simplification*.

In general, given edges $A \rightarrow B$ and $B \rightarrow C$, it is safe to replace $A \rightarrow B$ with $A \rightarrow C$ if the following hold:

1. Executing B does not change the value of any register or memory location.

2. Whenever control flows from A to B, the edge $B \to C$ must be taken (as opposed to any other edge $B \to D$).

For the first condition above, ensuring that the execution of a block B does not change the contents of any memory locationis difficult in general, and we resort to simple sufficient conditions, e.g., that B does not contain any *store* instructions. In particular, if B contains a single instruction, and that instruction is a branch, then B satisfies the first condition.

We can also use the following (weaker) condition in place of the second condition above:

$2(a)$. There is a predicate P such that: (i) P is true at exit from A; and (ii) P is false at the entry to each of B's successors except C.

If P is always true at the exit from A, then when control goes from A to B along the edge $A \to B$, P must be true at the beginning of B. If condition 1 is satisfied, then B does not change the value of P, so P must be also be true at the beginning of the block that is branched to from B. Among B's successors, P can only be true on entry to C, so control can only flow to C. Therefore conditions 1 and $2(a)$ imply condition 2.

Information about which predicates must be true or false at the entry to any basic block are derived from the guard predicates of conditional branches that transfer control to them, using a straightforward dataflow analysis that propagates truth values for predicate registers. This analysis is conceptually simply a straightforward application of constant propagation to predicate registers, and is not discussed further here.

Path simplification of our example CFG produces the CFG shown in Figure 10(b). While the number of blocks and edges is unchanged, the number of paths from B_1 to B_8 has decreased from six to two.

9. Experimental Results

We evaluated our ideas using a set of nine programs from the SPECint-2000 benchmark suite: *bzip2*, *crafty*, *gap*, *gzip*, *mcf*, *parser*, *twolf*, *vortex*, and *vpr*. The programs were compiled using Intel's *ecc* compiler version 5.0.1, at optimization level `-O3` together with profile feedback, and run on an HP i2000 workstation with a 733 MHz Intel Itanium processor with 1 GB of main memory, running Redhat Linux 7.1, kernel 2.4.3-12. We used statically linked binaries for our experiments, compiled with additional flags to instruct the linker to retain relocation information (relocation information is used by our particular implementation during disassembly, but is not fundamental to any of the algorithms described in this paper).

The baseline for our experiments is the set of control flow graphs obtained using the naive algorithm described in Section 3. Figure 11 illustrates the extent to which these control flow graphs could be simplified using the ideas described in this paper, showing the percentage reductions obtained, respectively, in the number of basic blocks, control flow edges, and instructions in the reverse engineered program. It can be seen that unpredication based on predicate analysis, by itself, is able to achieve fairly modest reductions in flow graph complexity: the number of basic blocks decreases by 3.4%–7.7% (5.2% on average);[3] the number of edges by 2.0%–5.0% (3.2% on average), and the number of instructions by 1.0%–2.4% (1.6% on average). The reason for these modest numbers is that for most of these programs, instruction scheduling has the effect of ordering instructions such that, even after predicate analysis, the predicate groups that we are able to construct are often not very large.

However, unscheduling is able to undo much of these effects of instruction scheduling. Thus, when unscheduling is added to predicate analysis based unpredication, there are significant improvements to the amounts of flow graph simplification that we are able to achieve. Compared to the results of naive reverse engineering, the number of basic blocks decreases by 8.6%–17.9% (13.9% on average); the number of edges by 6.0%–13.5% (average: 10.3%); and the number of instructions by 1.9%–4.6% (average: 3.3%).

Unspeculation is able to improve these results with varying degrees of success for each benchmark. Relative to the results of naive reverse engineering, the number of basic blocks decreases by 15.0%–25.6% (average: 18.9%), the number of edges by 11.1%–21.5% (average: 14.7%), and the number of instructions by 3.8%–10.9% (average: 5.6%). One of the reasons for the apparently small effect of unspeculation in some cases is that we consider statically linked binaries, and the library code we used did not have any speculation. Therefore, for small benchmarks such as *gzip* and

[3] All averages quote here are computed as geometric means.

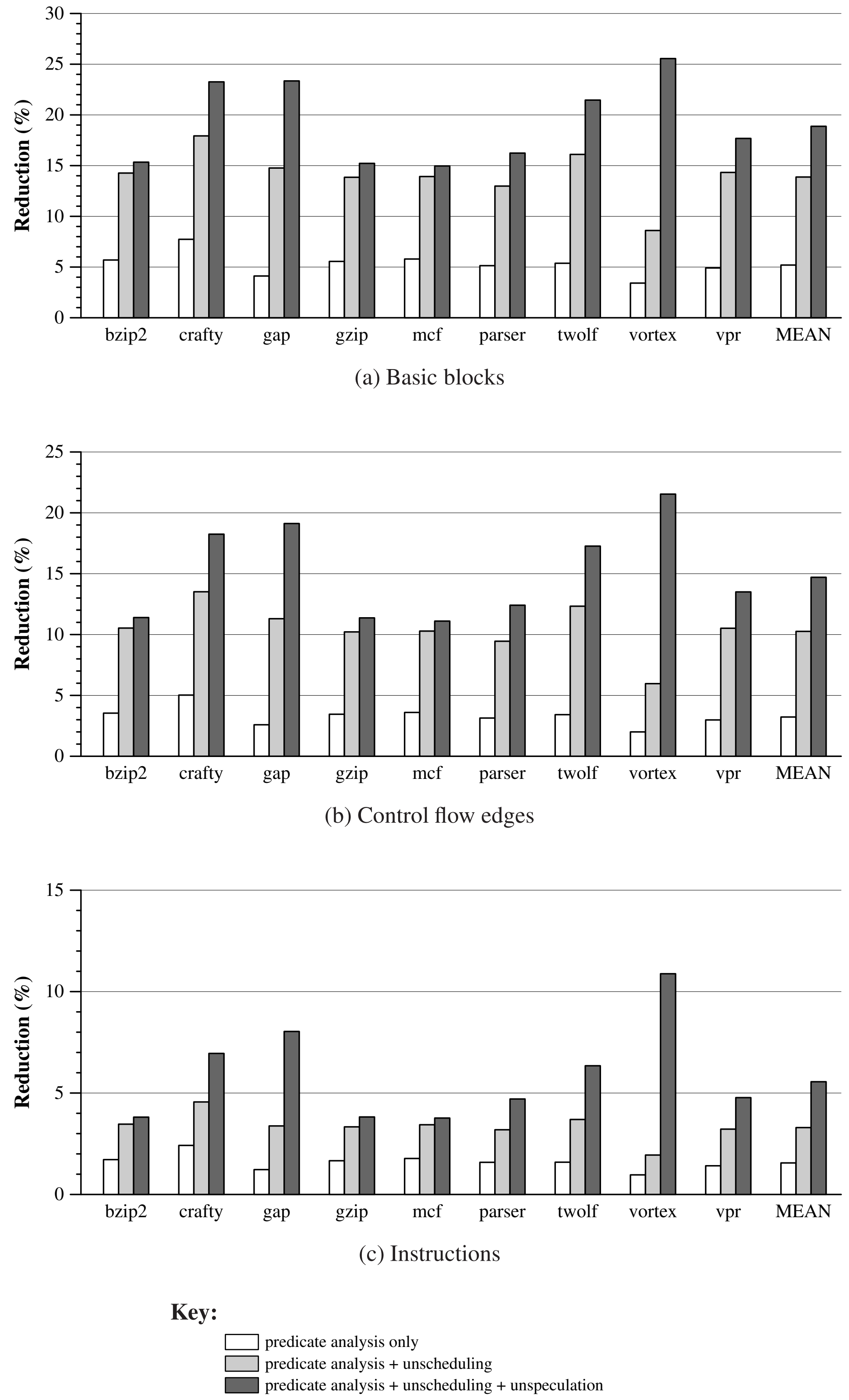


Figure 11. Effects of program analysis and transformations on the complexity of control flow graphs

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

mcf, unspeculation affects a relatively small portion of the code, whereas for large benchmarks like *gap* and *vortex*, unspeculation has a more noticeable effect. If we confine ourselves only to user code, we find that unspeculation is able to eliminate about 75% of the speculative loads and speculation checks, as shown in Table 1; this results in average reductions of 14.0% in the number of basic blocks, 12.7% in the number of control flow edges, and 6.8% in the number of instructions (all relative to the user code without unspeculation).

Overall, we accomplish significant improvements in the quality of reverse engineering: relative to a naive approach we obtain an average reduction of 18.9% in the number of basic blocks, 14.7% in the number of control flow edges, and 5.6% in the number of instructions, in the control flow graphs constructed from optimized Itanium executables.

10. Conclusions

EPIC architectures such as the Intel Itanium contain a number of architectural features, such as predication of instructions, explicit instruction-level parallelism, and speculative execution, that can significantly improve performance. However, aggressively optimizing a program to take advantage of these features can restructure the low-level structure of the code dramatically, making it difficult to reconstruct the original program logic via reverse engineering. This paper describes a number of techniques that can be used to undo many of the effects of such program transformations, so as to simplify the task of identifying the original program structure and reasoning about its behavior. Experimental results indicate that our ideas are able to effect significant reductions in the size and complexity of the control flow graphs obtained in the course of reverse engineering optimized Itanium executables.

References

[1] S. Burford. Reverse engineering Linux ELF binaries on the x86 platform, 2002. `www.linuxsa.org.au/meetings/reveng-0.2.pdf`.

[2] E. J. Byrne. Software reverse engineering: a case study. *Software—Practice and Experience*, 21(12):1349–1364, 1991.

[3] C. Cifuentes and K. J. Gough. Decompilation of binary programs. *Software—Practice and Experience*, 25(7):811–829, July 1995.

[4] C. Cifuentes and D. Simon. Procedural abstraction recovery from binary code. In *Proc. European Conference on Software Maintenance and Reengineering*, March 2000.

[5] C. Cifuentes and M. Van Emmerik. UQBT: Adaptable binary translation at low cost. *IEEE Computer*, 33(3):60–66, March 2000.

[6] C. Cifuentes and M. Van Emmerik. Recovery of jump table case statements from binary code. *Science of Computer Programming*, 40(2–3):171–188, July 2001.

[7] D. M. Gillies, D. R. Ju, R. Johnson, and M. Schlansker. Global predicate analysis and its application to register allocation. In *Proc. 29th Annual International Symposium on Microarchitecture*, pages 114–125, 1996.

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

[9] C. R. Hollander. *Decompilation of object programs*. PhD thesis, Stanford University, 1973.

[10] R. Johnson and M. Schlansker. Analysis techniques for predicated code. In *Proc. 29th Annual International Symposium on Microarchitecture*, pages 100–113, 1996.

[11] J. B. Kam and J. D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, January 1976.

[12] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.

[13] Á. Kiss, J. Jász, G. Lehotai, and T. Gyimóthy. Interprocedural static slicing of binary executables. In *Proc. 3rd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 118–127, September 2003.

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

[15] S. S. Liao, P. H. Wang, H. Wang, G. Hoflehner, D. Lavery, and J. P. Shen. Post-pass binary adaptation for software-based speculative precomputation. In *Proc. ACM SIGPLAN'02 Conference on Programming Language Design and Implementation (PLDI)*, June 2002.

[16] Todd A. Proebsting and Scott A. Watterson. Krakatoa: Decompilation in Java *(does bytecode reveal source?)*. In *Proceedings of The Third USENIX Conference on Object-Oriented Technologies and Systems*, pages 185–197. The USENIX Association, 1997.

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

[18] B. Schwarz, S. K. Debray, and G. R. Andrews. Disassembly of executable code revisited. In *Proc. IEEE 2002 Working Conference on Reverse Engineering (WCRE)*, pages 45–54, October 2002.

[19] J. W. Sias, W. W. Hwu, and D. I. August. Accurate and efficient predicate analysis with binary decision diagrams. In *Proc. of the 33rd Annual International Symposium on Microarchitecture*, pages 112–123, 2000.

[20] 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.

[21] N. Snavely, S. K. Debray, and G. R. Andrews. Unscheduling, unpredication, unspeculation: Reverse engineering itanium executables. In *Proc. 2003 IEEE Working Conference on Reverse Engineering*, November 2003.

[22] N. Snavely, S. K. Debray, and G. R. Andrews. Unspeculation. Technical report, Dept. of Computer Science, The University of Arizona, May 2003.

[23] T. Systä, K. Koskimies, and H. Müller. Shimba: an environment for reverse engineering Java software systems. *Software Practice & Experience*, 31(4):371–394, April 2001.

Author Biographies

Noah Snavely

Noah Snavely received his BS in computer science and mathematics from the University of Arizona in 2003. He is now a Ph.D student in computer science at the University of Washington, supported by a National Science Foundation Graduate Research Fellowship. His interests lie in low-level compiler optimizations and in computer graphics and vision.

Saumya Debray

Saumya Debray is Professor of Computer Science at The University of Arizona, where he has been a faculty member since August 1986. Prior to this, he received a B.Tech. (Hons.) degree in Electronics and Electrical Communications Engineering from the Indian Institute of Technology, Kharagpur, in 1981; and M.S. and Ph.D. degrees in computer science, from the State University of New York at Stony Brook, in 1983 and 1986 respectively. Saumya's research interests are primarily in the area of compilers and language implementation. His current research focuses on binary rewriting and its applications to reverse engineering, code optimization, code compression, and software security.

Gregory Andrews

Gregory R. Andrews received a B.S. in Mathematics from Stanford University in 1969 and a Ph.D. in Computer Science from the University of Washington in 1974. From 1974-79 he was an Assistant Professor at Cornell University. Since 1979 he has been at The University of Arizona, where he is currently Professor of Computer Science. From 1986-93 he chaired the department. Greg also worked in the CISE directorate at the National Science Foundation in 2003 and 2004, where he was the first division director of Computer and Network Systems. Greg received a distinguished teaching award in 1986 and a career distinguished teaching award in 2002. In 1998 he was named a Fellow of the Association for Computing Machinery (ACM).