Optimizing and Reverse Engineering Itanium Binaries [*]

Noah Snavely

EPIC (Explicitly Parallel Instruction Computing) architectures, such as the Intel IA-64 (Itanium), address common bottlenecks in modern architectures by supporting novel features such as explicit instruction-level parallelism, predicated instructions, and control and data speculation. While these features promise to make code more efficient, the fact that these new architectural features are visible to the programmer means that EPIC code is more difficult to generate and analyze than code for more traditional architectures. In this paper we discuss methods for dealing with Itanium code in a way that is less tied to the specific features of the Itanium architecture, using a system we have developed called the Itanium Link-Time Optimizer (ILTO). We also present new algorithms for generating efficient Itanium code and for reverse-engineering Itanium programs in the context of ILTO.

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

Contents

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), that boast features such as predicated instructions, explicit instruction-level parallelism, and control and data speculation. A predicated instruction is guarded by a Boolean source operand; the instruction is executed only if this guard evaluates to true. In addition, instruction-level parallelism is explicit: the compiler is responsible for collecting instructions into groups that will be executed in parallel. Control and data speculation refer to the existence of special instructions that have more freedom of movement during scheduling than normal instructions.

These features are intended to improve performance by increasing the number of instructions that can be executed in each cycle, in the case of predication and instruction-level parallelism, or by decreasing the amount of time the CPU spends waiting for memory fetches, in the case of speculation, but there are costs associated with these features. First, since these features are visible to the user, the burden is on the compiler to make effective use of them, and therefore new algorithms and techniques must be developed to generate good code. Second, heavily optimized EPIC code is difficult to analyze and transform with traditional algorithms, because these algorithms are typically unaware of features such as predication, and may produce incorrect or over-conservative results when applied to predicated code. Third, to someone who is trying to read and comprehend a program, optimized EPIC code may be inscrutable, because predication and speculation—particularly when combined with traditional optimizations such as instruction scheduling—tend to severely obfuscate the original program logic. In this paper we address, and present solutions to, each of these problems.

With regards to the first problem, predication is one of the most important features for the compiler to be aware of. In order to generate efficient code, a compiler must selectively eliminate conditional jumps in favor of predicated instructions that are conditionally executed. This process, known as *if-conversion*, must be carried out judiciously: if it is too aggressive, it leads to contention for system resources and a concomitant degradation in performance; if it is not aggressive enough, it results in insufficient instruction-level parallelism, which also leads to a loss in performance. An important question that must be addressed in this regard is: when in the compilation process should if-conversion be carried out? One option is to do if-conversion early in the code generation process, with subsequent analyses and optimizations working on predicated code. This is the approach taken by August *et al.* [3], who carry out aggressive if-conversion early, and subsequently perform partial reverse if-conversion during instruction scheduling. The advantage of such an approach is that the compiler can take full advantage of instruction predication in a variety of low-level optimizations. A disadvantage is that this exacerbates the second problem mentioned above: analyses and optimizations in the compiler backend may have to be reimplemented to cope with predication.

In this paper we describe a system we have developed, called the Itanium Link-Time Optimizer (ILTO), that uses the opposite strategy: machine dependent optimizations such as if-conversion are delayed until after most other optimizations have been carried out. The advantage here is that other analyses and optimizations do not have to be made predicate-aware. Our system is a *binary-rewriter*, so its input is a program that may already be heavily optimized. Therefore in addition to presenting a new algorithm for if-conversion, we also present algorithms for the dual problem of *unpredication*, or *reverse if-conversion*, that is, removing predication from code. In the context of both if-conversion and reverse if-conversion, it is useful to have knowledge of certain relationships between predicate registers to perform these tasks effectively. Therefore we also describe a simple algorithm for analyzing predicated code and computing these relationships.

With respect the third problem, that of program comprehension, we address the problems associated with speculation. Optimizations based on speculation can significantly change the structure of programs, speculation tends to make low-level code difficult to understand and analyze. In this paper we present a technique for undoing low-level optimizations based on speculation in order to expose the original structure of speculative programs.

The remainder of the paper is organized as follows. Section 2 gives background information on the Itanium architecture and on ILTO, our experimental optimization system. Section 3 describes how we analyze the use of predicate registers to compute what we call predicate disjointness and dominance sets. Section 4 describes the front-end of ILTO, which transforms raw Itanium code into an intermediate form. Section 5 describes the back-end of ILTO, in which the intermediate code is transformed back into efficient Itanium code using new algorithms, and shows the effectiveness of these algorithms experimentally. Section 6 discusses reverse engineering issues on the Itanium, in particular those related to speculation, and describes how we have used ILTO to reverse engineer Itanium binaries. Finally, Section 7 discusses related work, and Section 8 gives concluding remarks.

2 Overview

The work reported in this paper was carried out in the context of ILTO, a link-time optimizer we have developed for the Intel Itanium processor. This section summarizes relevant aspects of the Itanium architecture, including predicated instructions, instructions groups, bundles, and templates. We then give an overview of the organization of ILTO.

2.1 The Itanium Architecture

2.1.1 Explicit Parallelism and Predication

The Itanium contains multiple functional units and uses programmer specified instruction-level parallelism. Moreover, every instruction is *predicated*: It specifies a one-bit predicate register, and if the value of that register is true (1), then the instruction is executed; otherwise, the instruction usually has no effect. The Itanium has 64 predicate registers; register `p0` has 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*.

Predicate registers are set by compare instructions. There are three broad classes of compares: normal, unconditional, and parallel. A normal compare has four operands: two data operands that are compared, and two predicate registers that are assigned the result and its complement. An unconditional compare is like a normal compare, except that it clears both predicate-register operands before doing the data comparison and setting the results; moreover, the predicate registers are cleared even if the instruction is not executed because its guard is false. 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.

The compiler writer or assembly programmer expresses parallelism by forming what are called *instruction groups*. Each group is a sequence of instructions that do not contain register dependencies and hence that can potentially be issued in parallel. In particular, instructions in a group cannot in general contain read-after-write (RAW) or write-after-write (WAW) register dependencies. (Write-after-read dependencies are allowed in a group since the processor will ensure that the read occurs before the data is overwritten.) The programmer indicates the end of an instruction group by means of what are called *stop bits*.

Following is an example of a sequence of predicated instructions:

```
        cmp.eq p6,p7=r10,r11
   (p6)  ld8    r15=[r32]
   (p7)  ld8    r16=[r33] ;;
   (p6)  add    r15=r15,1
   (p7)  add    r16=r16,1 ;;
   (p6)  st8    [r32],r15
   (p7)  st8    [r33],r16
```

The first instruction is unguarded and always executed. It compares the contents of general registers `r10` and `r11`; if they are the same, predicate register `p6` is set to true and register `p7` is set to false; otherwise `p7` is set to true and `p6` is set to false. Because the values of `p6` and `p7` are complements of each other, exactly one set of load, add, store instructions will execute, depending on which of `p6` or `p7` is true. There are register dependencies between the add and load instructions, and between the store and add instructions, so stop bits—indicated by double semicolons `;;`—are placed after the pair of loads and the pair of adds.

The Itanium processor fetches instruction *bundles* that are 128 bits long (two words). Each bundle consists of three 41-bit instruction *slots* and a 5-bit *template*. The template specifies the kind of functional unit needed by each instruction—integer, memory, branch, etc.—and where stop bits are located. The processor views up to two bundles (six instructions) at a time and attempts to *disperse* all of them to functional units in parallel. An instruction can be dispersed when a functional unit is available; up to six instructions can be dispersed at the same time, but instructions are never dispersed out of order.

An instruction *issues* when it can be dispersed and when all the resources it requires (e.g., source registers) are available. A *split issue* occurs whenever an instruction does not issue at the same time as the previous instruction.

(Split issue leads to a delay of at least one clock cycle.) Stop bits always cause a split issue, because they indicate the presence of register dependencies. On the other hand, predication never causes a split issue.

2.1.2 Speculation

In addition to predication, the Itanium supports *speculation*. Speculation refers to the execution of instructions before it is known that it is necessary, or possibly even safe, to execute them, and is intended to give the compiler more freedom when scheduling instructions. On the Itanium speculation is expressed with two special types of load instructions: *control* speculative and *data* speculative loads. Control speculative loads may be moved past branch instructions on which they are dependent, while data speculative loads may be moved past potentially dependent store instructions. Speculation checks are also provided to either verify the success of a speculative load, or branch to recovery code in case of a failed speculative load. The benefit to using speculation is that potentially high-latency load instructions, when speculative, can be executed earlier than is otherwise possible. Speculation is discussed in more detail in Section 6.

To summarize, Itanium instructions are predicated, and they have to be placed into groups (demarcated by stop bits) and bundles (with associated templates). Using predicates wisely and scheduling instructions efficiently are thus keys to producing efficient code. The Itanium also supports speculative loads, which can be used to reduce the time the CPU spends waiting on memory.

2.2 ILTO: The Itanium Link-Time Optimizer

Our experimental infrastructure is a software system called ILTO (Itanium Link-Time Optimizer). ILTO has the same basic structure as PLTO, a link-time optimizer we have developed for the Intel IA-32 (Pentium) architecture [18]. In particular, ILTO reads in a binary object file, disassembles the code, carries out numerous analyses and code optimizations, performs if-conversion and code scheduling, and finally lays out code blocks and assembles a new binary. The place occupied by ILTO in the compilation process is illustrated in Figure 2.2.

For code analysis and optimization purposes, ILTO constructs a control flow graph (CFG) for each function in a program [1]. Control flow across function boundaries is represented using an *interprocedural control flow graph* (e.g., see [15]). It consists of the control flow graphs of all the functions in the program, together with edges representing calls and returns that connect the flow graphs of different functions. As shown in Figure 2, a function call is represented using a pair of blocks, a *call block* and a *return block*. There is a *call edge* from a call block to the entry block of the callee, with a corresponding *return edge* from the exit block of the callee to the return block. Indirect function calls are modeled using a special pseudo-function $\mathsf{F}_\perp$ that represents worst-case behaviors; e.g., it uses and defines all registers, writes to all memory locations, etc.

Disassembly and assembly are obviously architecture dependent. However, the representation of basic blocks, structure of the CFG, and—most importantly—the various analyses and optimizations are essentially the same as in PLTO. The special characteristics of the Itanium—such as predication, instruction groups, and bundles—are thrown away as the control flow graph is created. This lessened the time it took to develop ILTO, and more importantly it permits existing architecture-independent analyses and optimizations to be employed. However, it means that we have to deal with predication, stop bits, and bundling when scheduling and laying out code.

The transformation of the input binary executable into a normalized intermediate form constitutes the front-end of ILTO. The front-end prepares the binary to be analyzed and transformed in the optimization stage. Finally, in the back-end of ILTO normalized code is transformed back into an efficient stream of Itanium instructions, and the file is written back to disk. These stages are briefly described below:

1. *Front-end*

   (a) *Build Control Flow Graph.* Disassemble instruction bundles and build a control flow graph (CFG) with individual instructions. Eliminate dead code by doing a depth-first search from the entry point to mark reachable code.

   (b) *Predicate Analysis.* Compute predicate register relation sets, as described in Section 3.

   (c) *Unschedule Instructions.* Group together related instructions.

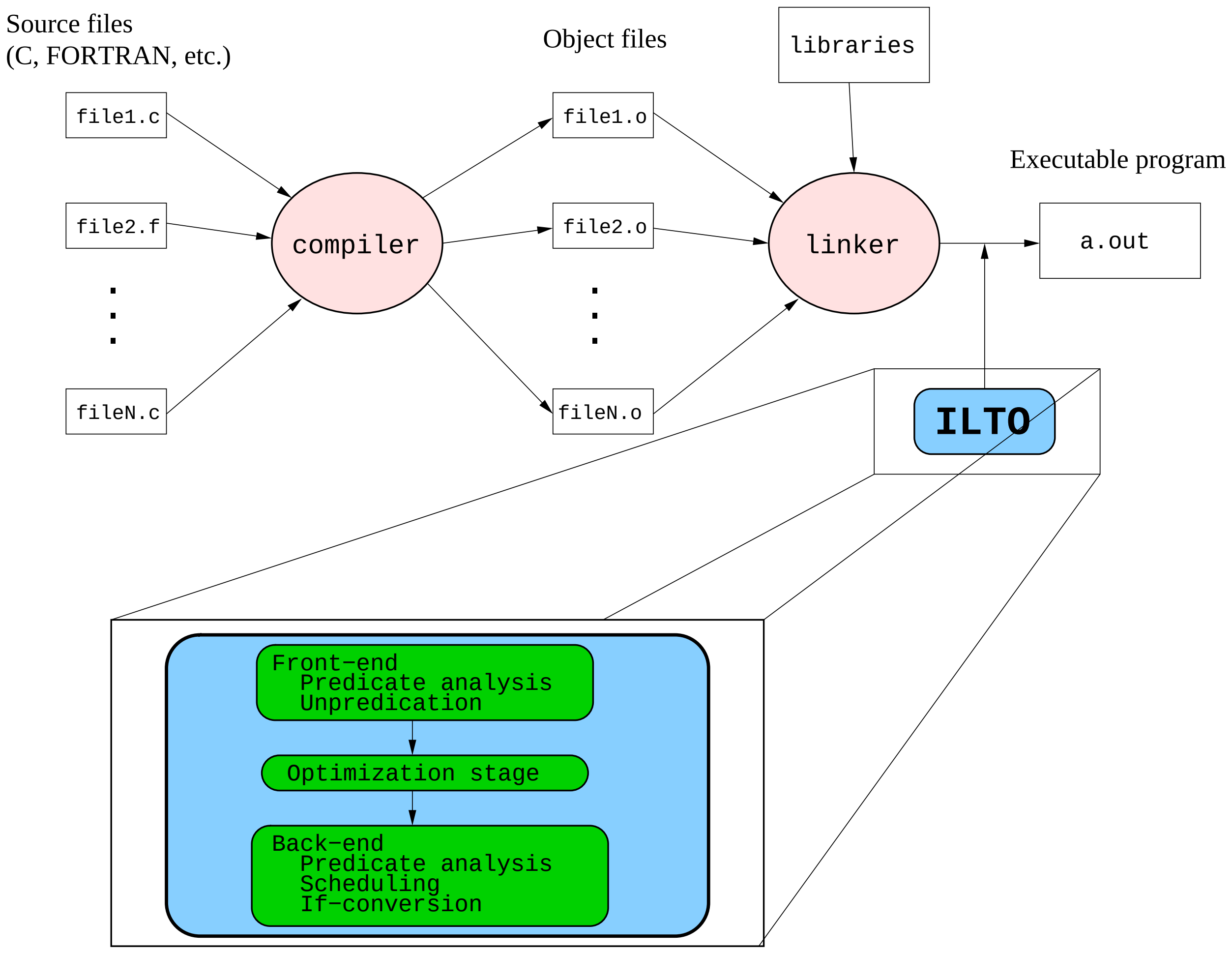


Figure 1: Compilation Model for Link-Time Optimization with ILTO

(d) *Unpredicate the CFG.* Remove predication from the instructions in the CFG by constructing explicit decision nodes.

2. *Optimization stage*

   (a) *Code Optimizations.* Analyze and optimize the code: liveness analysis, function inlining, constant propagation, etc. For this paper, this phase is not used, as discussed in the results section.

3. *Back-end*

   (a) *Scheduling and If-Conversion.* Form a schedule for each basic block and convert decision nodes to predicated instructions where possible. Group instructions into bundles.
   (b) *Predicate Analysis.* Recompute predicate register relation sets.
   (c) *Code Layout.* Layout and align the basic blocks, using edge profiles as a guide. (Edge profiles are generated during a training run on an instrumented version of the unpredicated CFG.)
   (d) *Global Bundle Check and Patch.* Iterate through the basic blocks to check the validity of instruction bundles and to repair them when needed.

Predicate analysis, used in both the front- and back-ends of ILTO, is described in the next section. After the discussion of predicate analysis, the front- and back-ends themselves are described in detail.

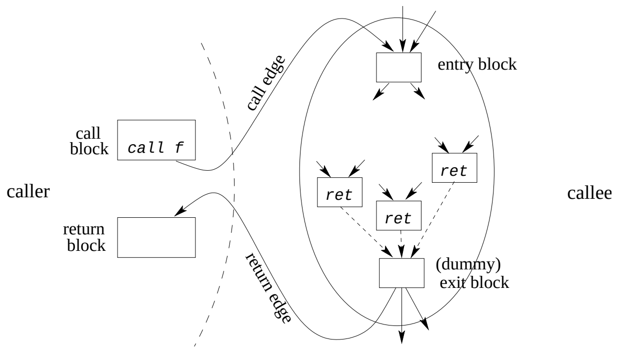


Figure 2: Representing function calls in the interprocedural control flow graph

3 Predicate Analysis

As mentioned in Section 2, predicate registers on the Itanium are written to by comparison instructions. Relationships that exist between predicate registers can be inferred at every point in the program by taking the semantics of these special instructions into account. In this section we will first discuss the predicate relationships that are of interest because of their use in analyzing predicate code. Next, we will describe the predicate analysis algorithm that we use to compute these relationships.

3.1 Predicate Relations

Given Booleans p and q, '$p \Rightarrow q$' denotes logical implication, i.e., $p \Rightarrow q \equiv (\neg p) \vee q$, while '$p \Leftrightarrow q$' denotes logical equivalence, i.e., $p \Leftrightarrow q \equiv (p \Rightarrow q) \wedge (q \Rightarrow p)$. We define the following notions of disjointness:

3.1.1 Disjointness

Definition 3.1 Booleans p and q are said to be *weakly disjoint* if $p \Rightarrow \neg q$. They are said to be *strongly disjoint* (or *complementary*) if $p \Leftrightarrow \neg q$. ▮

Note that both weak and strong disjointness are symmetric—e.g., if $p \Rightarrow \neg q$, then $q \Rightarrow \neg p$—so it is not necessary to specify directionality for either of them.

As an example, the following instruction sets predicate registers `p6` and `p7` to complementary values, depending on whether general register `r5` is less than register `r6`:

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

Immediately after this instruction, `p6` and `p7` are strongly disjoint, independent of their actual values. They remain strongly disjoint until some instruction (on some path) invalidates the relationship.

Suppose 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 strongly disjoint, even though their values might have changed. (If `p6` and `p7` were weakly disjoint before this instruction, they would also be weakly disjoint after it; if we knew nothing about their relation before the instruction, we would still know nothing.)

The weakly disjoint relationship most often arises due to instances of unconditional compare instructions. An example is

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

If `p8` is true, the semantics of this instruction are the same as a normal compare. However, an unconditional compare first clears both predicate operands, `p6` and `p7` above, and these remain cleared if the guard predicate is false. Thus, after this instruction, `p6` and `p7` are weakly disjoint: they cannot both be true but they might both be false.

```
initialize $weak\mathsf{IN}(B)$, $strong\mathsf{IN}(B)$, and $dom\mathsf{IN}(B)$ as described in the text;
$weak\mathsf{OUT}(B)$ = $weak\mathsf{IN}(B)$;
$strong\mathsf{OUT}(B)$ = $strong\mathsf{IN}(B)$;
$dom\mathsf{OUT}(B)$ = $dom\mathsf{IN}(B)$;
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;
  /* Assume $I$ has the form: ($pG$) compare-opcode $pA$,$pB$=data-operands */
  if $I$ is a normal compare instruction then
    if $I$ is unguarded, i.e., $pG$ == p0 then
      remove all pairs containing $pA$ or $pB$ from $weak\mathsf{OUT}(B)$, $strong\mathsf{OUT}(B)$, and $dom\mathsf{OUT}(B)$;
      add ($pA$, $pB$) to $weak\mathsf{OUT}(B)$ and $strong\mathsf{OUT}(B)$;
    else
      set $wasInWeak$ to true if ($pA$, $pB$) is in $weak\mathsf{OUT}(B)$ and to false otherwise;
      set $wasInStrong$ to true if ($pA$, $pB$) is in $strong\mathsf{OUT}(B)$ and to false otherwise;
      remove all pairs containing $pA$ or $pB$ from $weak\mathsf{OUT}(B)$, $strong\mathsf{OUT}(B)$, and $dom\mathsf{OUT}(B)$;
      if $wasInStrong$ then
        add ($pA$,$pB$) to $weak\mathsf{OUT}(B)$ and $strong\mathsf{OUT}(B)$;
      else if $wasInWeak$ or $pG == pA$ or $pG == pB$ then
        add ($pA$,$pB$) to $weak\mathsf{OUT}(B)$;
      else
        /* now no relations between $pA$ and $pB$ */
      end if
    end if
  else if $I$ is an unconditional compare instruction then
    remove all pairs containing $pA$ or $pB$ from $weak\mathsf{OUT}(B)$, $strong\mathsf{OUT}(B)$, and $dom\mathsf{OUT}(B)$;
    add ($pA$, $pB$) to $weak\mathsf{OUT}(B)$;
    add ($pG$, $pA$) and ($pG$, $pB$) to $dom\mathsf{OUT}(B)$;
    for all ($p$, $pG$) that are in $weak\mathsf{OUT}(B)$ do
      add ($p$, $pA$) and ($p$, $pB$) to $weak\mathsf{OUT}(B)$;
    end for
    for all ($p$, $pG$) that are in $dom\mathsf{OUT}(B)$ do
      add ($p$, $pA$) and ($p$, $pB$) to $dom\mathsf{OUT}(B)$;
    end for
  else /* $I$ is a parallel AND or OR compare instruction */
    remove all pairs containing $pA$ or $pB$ from $weak\mathsf{OUT}(B)$, $strong\mathsf{OUT}(B)$, and $dom\mathsf{OUT}(B)$;
  end if
end for
```

Figure 3: Computing Predicate Relation Sets for a Basic Block

3.1.2 Dominance

Another important predicate relationship that arises from the use of unconditional compares is dominance.

Definition 3.2 A boolean p is said to *dominate* a boolean q if $q \Rightarrow p$, i.e. if p must be true whenever q is true. ▮

For instance, after execution of the compare instruction

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

`p8` dominates both `p6` and `p7`, because the semantics of the unconditional compares guarantee that if `p8` is false, both `p6` and `p7` will also be false. Note that while the disjointness relations are symmetric, dominance is not. In fact, the dominance relation creates a partial-ordering of predicate registers at every program point. For each register p, at any given program point there is a unique maximal dominance chain $p \Rightarrow p_1 \Rightarrow p_2 \Rightarrow \ldots \Rightarrow p_k$. Recall that on the Itanium register `p0` is hardwired to `true`, so `p0` dominates every predicate register. Therefore `p0` terminates any maximal dominance chain. Dominance chains are used during unpredication of the control flow graph, described in Section 4.

3.2 Predicate Analysis Algorithm

In order to do effective unpredication, if-conversion, and instruction scheduling (see Sections 4 and 5), we need to know—at each instruction—how predicate registers are related to each other. In particular, for a given register, which other register is strongly disjoint from it, which other registers are weakly disjoint from it, and which registers dominate or are dominated by it? (There can be at most one register that is strongly disjoint, but there could be several that are weakly disjoint, as well as several that dominate.) The predicate analysis phases in the ILTO system compute this information for the start and end of each basic block in a program, as described below. (It is straightforward to propagate information from the start of a basic block to instructions in the block.)

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. We consider three kinds of such sets at each basic block B:

Definition 3.3 Set $weak\mathsf{IN}(B)$ is the set of pairs of weakly disjoint predicates at the entry to block B, and $weak\mathsf{OUT}(B)$ is the set of pairs of weakly disjoint predicates at the exit from block B. Similarly, $strong\mathsf{IN}(B)$ is the set of pairs of strongly disjoint predicates at the entry to block B, and $strong\mathsf{OUT}(B)$ is the set of pairs of strongly disjoint predicates at the exit from B. Finally, $dom\mathsf{IN}(B)$ is the set of "dominance" pairs (p, q), where p dominates q, at the entry to block B, and $dom\mathsf{OUT}(B)$ is the set of dominance pairs 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 six sets are computed.

1. The dataflow information at the exit from a basic block B is obtained, as usual, by taking the dataflow information entering B and propagating it through B. In particular, $weak\mathsf{OUT}(B)$ is a function of $weak\mathsf{IN}(B)$ and the instructions in B, and similarly $strong\mathsf{OUT}(B)$ is a function of $strong\mathsf{IN}(B)$ and the instructions in B, and $dom\mathsf{OUT}(B)$ is a function of $dom\mathsf{IN}(B)$ and the instructions in B.

2. Determining predicate 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:
   $$weak\mathsf{IN}(B_0) = strong\mathsf{IN}(B_0) = dom\mathsf{IN}(B_0) = \emptyset.$$

   (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 predicate 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:
   $$weak\mathsf{IN}(B) = \mathsf{FnOut}_f(weak\mathsf{OUT}(B')),$$
   $$strong\mathsf{IN}(B) = \mathsf{FnOut}_f(strong\mathsf{OUT}(B')), \text{ and}$$
   $$dom\mathsf{IN}(B) = \mathsf{FnOut}_f(dom\mathsf{OUT}(B')).$$

   (c) Otherwise, it consists of the predicate relations that hold at the exit from each of B's predecessors, and so are guaranteed to hold at entry to B:
   $$weak\mathsf{IN}(B) = \bigcap_{P \in preds(B)} weak\mathsf{OUT}(P),$$
   $$strong\mathsf{IN}(B) = \bigcap_{P \in preds(B)} strong\mathsf{OUT}(P), \text{ and}$$
   $$dom\mathsf{IN}(B) = \bigcap_{P \in preds(B)} dom\mathsf{OUT}(P).$$

Figure 3 gives the algorithm for computing $weak\mathsf{OUT}(B)$, $strong\mathsf{OUT}(B)$, and $dom\mathsf{OUT}(B)$ from $weak\mathsf{IN}(B)$, $strong\mathsf{IN}(B)$, and $dom\mathsf{IN}(B)$. There are several cases to consider, but the details are straightforward applications of the kinds of reasoning illustrated in the examples at the start of this section. For example, a normal comparison makes its predicate-register operands strongly disjoint and hence also weakly disjoint; thus, the pair of operands gets added to both the strong and weak output sets. The unconditional compare instruction has the most complex effect, because it clears both predicate-register operands before conditionally setting one of them. A parallel compare instruction has the simplest effect with respect to predicate disjointness because it either does nothing or modifies both predicate-register operands, and hence it destroys any disjointness relationship that might have existed for either predicate register.

We solve the dataflow equations given above by starting with the initial values

$$weak\mathsf{IN}(B) = strong\mathsf{IN}(B) = dom\mathsf{IN}(B) = \emptyset$$

$$weak\mathsf{OUT}(B) = strong\mathsf{OUT}(B) = dom\mathsf{OUT}(B) = \emptyset$$

for all basic blocks B in a function, and then computing a fixpoint by iteratively applying the equations above until there is no change to any of these sets.

In case 2(b) of the dataflow equations above, $\mathsf{FnOut}_f(S)$ denotes the effect of the function call f on the predicate relations at the call site. A simple conservative estimate for intra-procedural analyses is to assume that nothing is known about predicate relationships at the return from a function call. We can do better, however, by identifying for each function f, the set $\mathsf{Unchg}(f)$ of predicate registers whose values will not be affected by a call to f. We proceed 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 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 is to propagate $\mathsf{Unchg}(f)$ values over the call graph of the program and iterate to a fixpoint. This is what we have implemented.

It is relatively straightforward to extend these equations to do inter-procedural analysis. At this time, we have extended the analysis described above into a simple context-insensitive inter-procedural algorithm, and we are looking into a context-sensitive inter-procedural version.

4 The Front-end of ILTO

The first purpose of the front-end is to first read in an Itanium executable, disassemble it, and construct the CFG to represent the control flow of the input program. This results in a stream of Itanium instructions in which special features of the Itanium are still visible. In particular, instructions are still grouped into bundles and instruction groups, and are still predicated. The second purpose of the front-end is to remove the traces of these features from the code. First we unbundle and ungroup instructions by removing stop bits. We then remove predication from the code by replacing guard predicates by decision nodes and adding new basic blocks and edges to the CFG. This process is known as *unpredication* or *reverse if-conversion*.

One way to unpredicate the instruction stream is to create a new basic block for every predicated (non-branch) instruction in the program. While correct and simple, this method would create huge number of basic blocks. The problem is that it ignores any relationships that exist between the predicates of nearby instructions, which can help to group instructions together during unpredication. Therefore, instead of making single instructions the basic units of unpredication, we can often take account of predicate relations to simplify the resulting CFG. Having a less-complicated CFG simplifies later analyses and makes it easier to produce efficient code later on. However, before we can take advantage of relationships between predicate registers during unpredication, there are two problems that must be solved. First, predicate relations are not explicit in Itanium code; they are implicitly created by a stream of one or more instructions that write to predicate registers. These relationships must be computed using predicate analysis, which was described in the previous section. Second, instructions with related guard predicates may not be adjacent, and so it may be difficult to combine related instructions into the same CFG node during unpredication. To alleviate this problem, prior to unpredication related instructions are physically grouped together in a phase called *unscheduling*.

4.1 Unscheduling

Aggressive scheduling permutes instructions within basic blocks. Assuming that the scheduler is semantics-preserving, the permutation it produces has the same meaning as the original code, but it may be more difficult to analyze because scheduling can destroy the correspondence between physical relations and semantic relations that often exist in unoptimized code. In particular, predicate groups created early on during compilation may be split up by unrelated instructions during scheduling. Consider the code

```
         mov r4 = r5
    (p6) mov r1 = r2
    (p6) add r2 = 8,r2
         sub r3 = 8,r1
```

This fragment contains two instructions predicated on `p6`. Since they are adjacent, it is easy to see that they can be combined into a single node during unpredication as follows:

```
         mov r4 = r5
    (p6) br.cond Label1
         br Label2

  Label1:
         /* Grouped instructions */
         mov r1 = r2
         add r2 = 8,r2

  Label2:
         sub r3 = 8,r1
```

However, suppose that the dependencies between these instructions are such that the compiler could have separated the two predicated instructions during scheduling, for instance:

```
    (p6) mov r1 = r2
         sub r3 = 8,r1
    (p6) add r2 = 8,r2
         mov r4 = r5
```

In this case it is not as clear that the `Instr2` and `Instr3` can be grouped in the same block during unpredication. If we fail to recognize the relationship between these two instructions, the unpredicated fragment will look like this:

```
    (p6) br.cond Label1
         br Label2

  Label1:
         /* First predicated instruction */
         mov r1 = r2
```

```
Label2:
       sub r3 = 8,r1
  (p6) br.cond Label3
       br.cond Label4

Label3:
       /* Second predicated instruction */
       add r2 = 8,r2

Label4:
       mov r4 = r5
```

This unpredicated fragment contains twice as many blocks (four) as the unpredicated fragment for which the two predicated instructions were originally adjacent, and contains two paths that are impossible to execute. This may impact analyses that occur later on in ILTO.

The goal of unscheduling is to group together related instructions that may have been separated during the compiler's instruction scheduling phase. To make this notion precise, we first define the basic unit of unpredication:

Definition 4.1 A *predicate group* is a maximal sequence of consecutive predicated instructions $(\mathtt{p}_1)\mathtt{I}_1, (\mathtt{p}_2)\mathtt{I}_2, \ldots, (\mathtt{p}_n)\mathtt{I}_n$, all in the same basic block, such that $\mathtt{p}_1, \ldots, \mathtt{p}_n$ are related predicates. ▮

(The precise relations between predicates in a predicate group will be described in the next section.) Note that predicate groups are not necessarily sequences of guarded instructions. A sequence of unguarded instructions (instructions whose guard predicate is `p0`) also form a predicate group, since each instruction in the sequence is guarded by the same predicate.

The unscheduling algorithm we present seeks to permute instructions within a basic block so as to minimize the number of predicate groups in that block. It does so by *merging* predicate groups whenever possible. Two predicate groups A and B can be merged if:

1. Each of the predicates that appear in A is related to each of the predicates that appear in B.
2. A and B can be moved next to each other.

A predicate group has some freedom of movement: a predicate group A can move past an adjacent group B as long as no dependencies exist between the instructions in A and the instructions in B. Assuming all other groups remain in place, an instruction group can occupy a range of positions whose boundaries are either dependent instruction groups or the boundaries of the basic block containing that group. This range extends both forwards (with the direction of control-flow) and backwards (against the direction of control-flow). Our unscheduling algorithm consists of two stages: first it finds the forward range of each predicate group, and attempts to find another group in that range with which the first can be merged. It then does the same for the backward range. The forward range scanning algorithm is described in more detail in figure 4; the backward range scan is completely analogous.

The scan must be done in both directions because the process of merging two groups is sometimes 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
       sub r3 = 8,r1
  (p6) add r2 = 8,r2
       mov r4 = r5
```

The first predicate group (made up of the first instruction) cannot be moved down to meet and merge with the second predicate group (made up of the third instruction), since there is a dependent instruction in the way. However, nothing prevents the second group from being moved up to meet the first.

```
/* Scan of forward range of predicate groups */
for each basic block $B$ do
  for each predicate group $G$ (in reverse order) do
    if $G$ is the last group in $B$ then
      continue
    endif
    for each predicate group $G'$ following $G$ do
      if $G$ and $G'$ can be merged then
        disconnect $G$
        insert $G$ immediately before $G'$
        merge $G$ and $G'$ into a single predicate group
        break
      else if some instruction in $G'$ is dependent on an instruction between $G$ and $G'$ then
        break
      else if $G'$ is the last predicate group $B$ then
        break
      endif
    endfor
  endfor
endfor
```

Figure 4: The Basic Unscheduling Algorithm (Forward Pass)

4.2 Unpredication

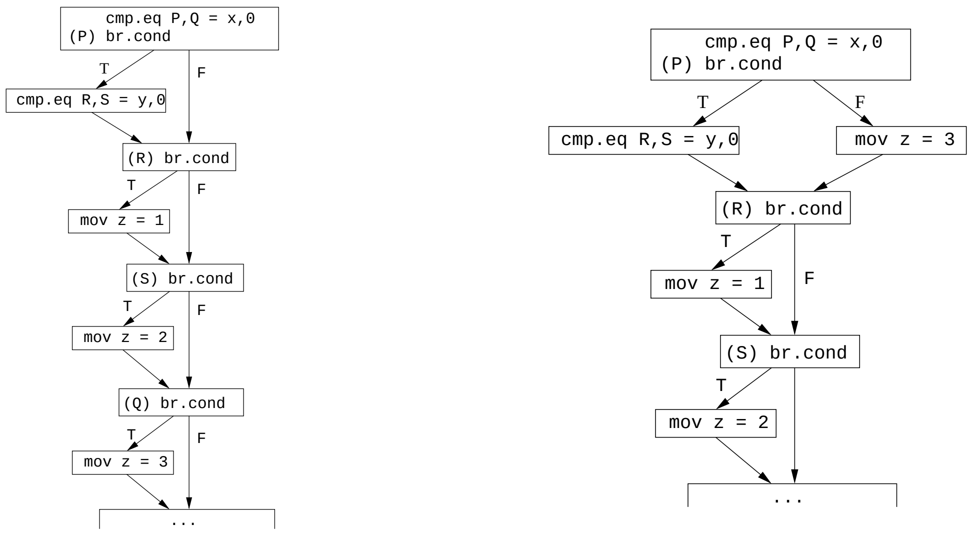
Unpredication, also called reverse if-conversion, is the process of replacing guard predicates with decision nodes and supporting control-flow structure. The basic algorithm we present to do this operates on predicate groups, and has two major steps: (i) find the predicate groups and (ii) for each predicate group G, disconnect each instruction in G from the CFG and insert it into a new block; afterwards adjust edges between blocks as needed. We defined predicate groups in the previous section, but did not specify exactly what relations constitute a predicate group. The core algorithm is independent of this definition, but the details are highly dependent. We will now give three definitions of a predicate group and show how the unpredication algorithm must change in response. Also, the effectiveness of unpredication is related to how rich the predicate group definition is – the relationships that are conserved upon unpredication are the same as the relationships captured by a predicate group. We will use a running example,

```
    cmp.eq p,q = x,0
(p) cmp.eq r,s = y,0
(r) mov z = 1
(s) mov z = 2
(q) mov z = 3
```

to demonstrate this.

The simplest way to define a predicate group is as a consecutive sequence of instructions guarded by the same predicate; we will call this a *simple predicate group.* Finding all the simple predicate groups in the program is straightforward, as is creating the control flow necessary to remove the predication. However, using this definition as the basis for our algorithm means that all relations implicit in the guard predicates of instructions—except for the identity relation—will be lost upon unpredication. This can be seen in Figure 5(a), which shows the results of unpredicating our example using simple predicate groups.

A smarter way to define a predicate group uses the complementary relation: a *complementary predicate group* is a consecutive sequence of instructions each guarded by either p or q, where p and q are complementary predicates. Finding complementary predicate groups is also easy, since predicate relation analysis gives us complementarity information at every program point. To unpredicate a complementary predicate group, we create two new blocks: a *true* block for the instructions predicated on p and a *false* block for the instructions predicated on q. Notice that in our running example p and q are complementary after the first instruction, but instructions predicated on p and q are



(a) Simple algorithm (b) Complement-aware algorithm

Figure 5: CFG After Unpredication

not adjacent. However, unscheduling will merge them into a single complementary predicate group. The resulting control-flow graph resulting from unpredicating our example using complementary predicate groups is shown in Figure 5(b). This algorithm preserves the complementary relationship between predicate registers by expressing it with an if-then-else control-flow structure, dominance information is lost. In this case, the fact that the instructions `mov z = 1` and `mov z = 2` can only be executed if `p` is true is not reflected in the structure of the control-flow graph.

Including the dominance relation in the definition of a predicate group makes the definition more complex. While a predicate typically has at most one complementary predicate at any given time, a predicate can be dominated by (or dominate) many other predicates. Recall that dominance defines a partial ordering of the predicate registers, so for a register `p`, there is a unique maximal dominance chain $p \Rightarrow p_1 \Rightarrow p_2 \Rightarrow \ldots \Rightarrow p_k$. This suggests a generalization of the model of predication used on the Itanium. Suppose we allow instructions to be guarded by several predicate registers, designated by $(p_1, p_2, \ldots, p_n)$, which are conjunctive; i.e., an instruction is executed if and only if each of its guard predicates is true. Then if an instruction I has guard predicate `p` and the dominance chain $p \Rightarrow p_1 \Rightarrow p_2 \Rightarrow \ldots \Rightarrow p_k$ exists at I, we can make dominance explicit by changing the instruction (p) `I` to the equivalent instruction $(p, p_1, \ldots, p_k)$ `I`. We will call this new model the *multi-predicate* model. For instance, under the multi-predicate model our example would look like this:

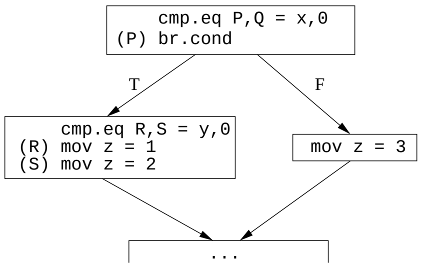
```
      cmp.eq p,q = x,0
  (p) cmp.eq r,s = y,0
(r,p) mov z = 1
(s,p) mov z = 2
  (q) mov z = 3
```

Note that if we use the multi-predicate model then we no longer need to make the second compare instruction unconditional—the same effect is achieved by guarding two of the move instructions with multiple predicates.

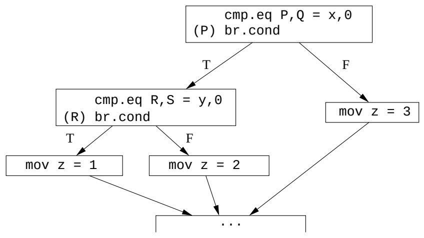
Now we can construct a definition of predicate group that takes both dominance and complementarity into account: a *full* predicate group is a sequence of consecutive instructions whose *most dominant* guard predicate is either `p` or `q`, where `p` and `q` are complementary predicates.

If each instruction is predicated on a single guard predicate then a full predicate group reduces to a complementary predicate group, for we have already described the unpredication algorithm. How then must the unpredication algorithm change to take advantage full predicate groups? We simply apply the algorithm multiple times. Each time the most dominant predicate guarding an instruction is removed, until none remain.

To see how this works, let us run the algorithm on the multi-predicate version of our example, and see how the code is changed after each pass. At first, there is a single predicate group in the block consisting of the three instructions



(a) CFG after first pass



(b) CFG after second pass

Figure 6: Two Passes of the Iterative Unpredication Algorithm

predicated on `p` and the one predicated on `p`'s complement, `q`. The first pass, will split these four instructions into two blocks as shown in figure 6(a).

The three instructions that had been predicated on `p` have been put in the `then` block, and `p` has been removed from their predicate lists. The instruction that had been predicated on `q` has been moved into the `else` block, and `q` has been removed from its predicate list. Before this pass, predicates `r` and `s` had been *conditionally complementary* on `p`; that is, if `p` is true, then `r` and `s` must be complementary predicates. After the pass, the instructions predicated on `r` and `s` exist in their own specialized path where the relation between them is stronger. In other words, `p` must be true when these instructions are executed, because the block containing them is only reached when `p` is true; hence the condition on which `r` and `s` are complementary is always satisfied.

On the next pass of the unpredication algorithm, the newly complementary instructions are separated into disjoint paths, as shown in figure 6(b). At this point, only conditional branches are predicated, so the algorithm terminates. The final control-flow graph explicitly expresses all the important information latent in the code containing predicated lists. The only remaining question is how to convert singly-predicated code into code that uses predicate lists. This can be done by computing dominance chains at each program point using predicate analysis. Note that the implementation omits `p0` from the end of computed dominance chains, since it is useful to consider instructions predicated on `p0` to be unpredicated (otherwise an iterative unpredication algorithm will never terminate!).

4.3 Edge simplification

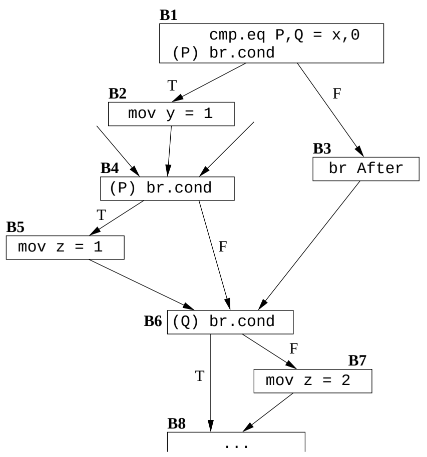
So far we have concentrated on preserving predicate relationships within a basic block; relationships that extend across basic-block boundaries are not necessarily exploited. For instance, consider the code:

```
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
```

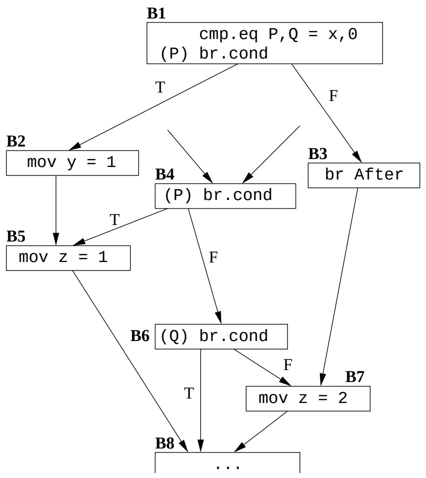
This fragment consists of three blocks, each of which will be unpredicated separately. Unpredication produces the control-flow graph shown in Figure 7(a).

In this control-flow graph there are multiple paths that can never be taken. For instance, it is impossible that blocks **B2**, **B4**, **B6** are executed in that order, because **B2** can only be reached if `p` is true – hence, if **B4** is reached from **B2**, the branch in **B4** must always be taken. So, nothing prevents us from redirecting the edge from **B2** to **B4** to instead point to **B5**. We call such a redirection an *edge simplification*. When is it possible to do such edge simplification? More formally, given a path from block A to block B to block C, under what conditions is it safe to replace the edge $A \to B$ be with the edge $A \to C$?

The conditions are twofold:



(a) Unpredicated CFG before edge simplification



(b) Unpredicated CFG after edge simplification

Figure 7: An example of edge simplification

1. Executing B does not change the value of any register or memory location; therefore the only program state that B can possibly change is the program counter.
2. Whenever control flows from A to B, the edge $B \to C$ must be taken (as opposed to any other edge $B \to D$).

If a block B contains a single instruction, and that instruction is a branch, then B satisfies the first condition. Checking that a block satisfies the second requires more analysis. To prove that control must flow in a certain direction along an execution path we must show that it is impossible for control to flow any other way. Our method for showing that a path is impossible is to find some predicate register that must be both true and false at some point along the path.

4.3.1 Dominating Predicates

Definition 4.2 A predicate p *pre-dominates* a basic block B if p must be true on entry to B. Similarly, p *post-dominates* B if p must be true at the end of B. ▮

Definition 4.3 A predicate p *pre-antidominates* a basic block B if p must be false on entry to B. Similarly, p *post-antidominates* B if p must be false at the end of B. ▮

These relationships usually arise from the guard predicates on branch instructions. Suppose that a block ends with a branch predicated on p (with complement q), and let A be the target of the branch and B be the fall-through block. Then p pre-dominates block A and pre-antidominates block B; conversely q pre-antidominates block A and pre-dominates block B. If no instruction in A writes to p or q then the relations are preserved through the block: P will post-dominate A and Q will post-antidominate A. To compute dominator sets we use a simple dataflow analysis.

Using the pre- and post- dominance, we can replace the second of the pair of edge-simplification conditions with a weaker condition:

$2(a)$. There exists a predicate p such that p post-dominates A and pre-antidominates each of B's successors except C.

If p post-dominates A then when control flows from 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 must flow to C. Therefore condition $2(a)$ implies condition 2.

Edge simplification of our example CFG produces the CFG shown in Figure 7(b). While the number of blocks and edges is unchanged, the number of paths from **B1** to **B8** has decreased from six to two.

5 The Back-end of ILTO

The back-end of ILTO is the complement to the front-end: its purpose is to transform the code from its intermediate form to a form that makes use of the special features of the Itanium in order improve efficiency. The most important stages of the back-end are *instruction scheduling*, in which we attempt to order Itanium instructions in an efficient way, and *if-conversion*, in which selected control flow structures are partially or fully replaced by predicated code. These two stages take place in parallel, as described below. After scheduling and if-conversion takes place, the code is laid out and reassembled, and the binary is written to disk. These steps are described below, with particular emphasis on scheduling and if-conversion. Afterwards we show the results of experimentally evaluating the performance impact of the transformations described in this section, and conclude that our approach does not degrade the performance of the input binary.

5.1 Instruction Scheduling

When scheduling instructions on the Itanium, there are two primary concerns. The first concern is to hide the latencies of instructions as much as possible. The latency of an instruction is the number of cycles that elapse between the time an instruction is executed and the time its results are ready. For instance, while addition of two integers takes a single cycle on the Itanium, adding two floating point numbers can take up to five cycles. Similarly, a load instruction can take anywhere from two to twenty-one cycles (depending on which level of cache is hit). If during execution an instruction attempts to use a register that is not yet available because a high-latency instruction has just written to it, then the processor stalls until the result is ready. Therefore, our scheduler attempts to schedule instructions so as to minimize the time that the processor needs to stall while awaiting the completion of a computation. To achieve this goal, we use a conventional list scheduling algorithm developed by Gibbons and Muchnick [7].

The second concern of our instruction scheduler is to exploit the instruction-level parallelism capabilities of the Itanium architecture by bundling instructions intelligently. As mentioned in Section 2, the Itanium can execute up to six instructions at once, but has only two arithmetic units, two memory units, two floating point units, and three branch units. Therefore in long sequences of code with no branches, instructions must be carefully bundled in order to maximize the throughput of a program. In order to make the most effective use of resources when bundling instructions, predicate analysis must be used to accurately compute dependencies between instructions. The reason for this is that two instructions may appear to be dependent (for instance they might write to the same register), but may in fact be independent if their predicates cannot both be true at the same time. For instance, consider the following code fragment:

```
     cmp.eq p6,p7=x,0 ;;
(p6) cmp.eq.unc p8,p9=y,0 ;;
(p7) mov z=2
(p8) mov z=0
(p9) mov z=1
```

Notice that even though the last three instructions all write to the same variable, `z`, the two compare instructions guarantee that exactly one register out of `p7`, `p8`, and `p9` will be true when the three moves are executed. We can determine this using predicate analysis, and therefore our bundler can schedule these three instructions in the same instruction group. Our instruction bundling algorithm is similar to one described in [9], but we augmented it to handle several special cases.

5.2 If-Conversion

If-conversion is the process of replacing explicit control transfers in code by predicated instructions that are executed conditionally depending on the value of a Boolean source operand [2]. It can improve performance in a number of different ways. First, it can eliminate difficult-to-predict branches and reduce branch misprediction rates [4]. Second, it can increase instruction-level parallelism. Finally, by allowing the producer of a value to be moved to an earlier point in the instruction stream, if-conversion can be used to hide instruction latencies.

Figure 8 gives an outline of our if-conversion algorithm. The basic idea is simple: For each basic block in a function, we first schedule the instructions in the block, then we try to use if-conversion to improve the code for that block. This employs the predicate disjointness sets described in the previous section and is done as follows:

```
for each basic block $B$ in the function do
  1. schedule $B$ as described above;
  2. sort the successors of $B$ in decreasing order of execution frequency;
  3. for each successor $S$ of $B$ do
       if $S$ has more than one predecessor continue;
       for each nop $N$ in $B$ do      /* Eliminate no-ops in $B$ if possible */
         if there is an instruction $I$ in $S$ that can replace $N$ without affecting any
                   dependencies or adding stop bits then
           remove $I$ from $S$;
           replace $N$ with an appropriately predicated version of $I$;
         endif
       end for
       /* Eliminate branch instructions in $B$ if possible and profitable */
       if $(a)$ $S$ is if-convertible into $B$; and $(b)$ there is a branch instruction $J$ in $B$ that
               can be eliminated by fully if-converting $S$ into $B$; and $(c)$ the number of
               groups in $S$ is less than a fixed [architecture-dependent] threshold then
         replace each instruction $K$ in $S$ by an appropriately predicated version of $K$ in $B$;
         delete the branch instruction $J$
         delete the basic block $S$
       end if
     end for
end for
```

Figure 8: The Basic If-Conversion Algorithm

1. We attempt to replace `nop`s in the block by useful instructions from its successor blocks.
2. If a block ends in a conditional branch, and it is profitable and possible to eliminate this branch, we replace the conditional branch by appropriately predicated instructions from the block's successors.

In this context, given a basic block B and a successor B' of B, we say that B' is *if-convertible into* B if every instruction in B' can be if-converted into a predicated version that can then be inserted at the end of B, prior to any branch instruction at the end of B, without altering any use-definition relationships between any pair of instructions.

A few aspects of this algorithm that deserve comment. First, when processing a basic block B and considering a successor block from which to if-convert instructions into B, we do not consider any successor S that has more than one predecessor. The reason for this is that if S has multiple predecessors, then each instruction moved out of S would have to be replicated in the predecessors of S. This would result in code growth, and it would complicate the if-conversion algorithm because it would be necessary to ensure that such code replication preserves correctness. In principle we could clone the block S in such circumstances to create a block with a single predecessor, which can then be processed as described; however, our implementation does not currently do this.

Second, when considering whether to use if-conversion to eliminate a branch instruction at the end of a block B, we want to make sure that this does not introduce so many predicated instructions into B that the cost of executing these instructions exceeds the cost of the original branch instruction they replaced. We do this using an architecture-dependent threshold that models the cost of executing a branch instruction: if the number of predicated instruction groups being introduced into B is less than this threshold, it is deemed profitable to eliminate the branch instruction. The reason we first attempt to use instructions from S to eliminate no-ops in B before attempting to eliminate branch instructions in B is that the number of instructions in S may initially exceed this threshold, but by pulling out instructions from S to replace no-ops in B, we may be able to reduce the number of instructions in S to below the threshold, thereby allowing the branch instruction in B to be eliminated.

Finally, an aspect of the overall if-conversion process that is not discussed in Figure 8 is that it is sometimes necessary to find a free predicate register. Consider the following code fragment:

```
cmp.eq p6,p0=r14,r15 ;;
```

```
      (p6) br.cond L1
           mov r14=0
           br.few L2 ;;
   L1:     mov r14=1 ;;
   L2:     add r15=r14,2
```

We would like to convert this to a single predicated block, e.g.:

```
           cmp.eq p6,p7=r14,r15 ;;
      (p6) mov r14=0
      (p7) mov r14=1 ;;
           add r15=r14,2
```

However, since the compare instruction that sets register `p6` in the original code discards the complement of `p6`, [1] we must find a predicate register to hold the complement. This register p must be free at the compare instruction and must not be defined on any path from the compare to the instruction(s) whose predicated version would use the complement of `p6`. If there are multiple compare instructions that set the guard predicate of the branch register (i.e., different paths to the branch contain different compare instructions), then p must not be defined on any path from any of the compares to the instructions that would use p. Our implementation currently uses a simple conservative approximation for this: If a predicate register p is not defined or used by a function f or any function reachable from f, and if p is saved and restored at entry to and exit from f, then p can safely be used for this purpose within f.

5.3 Code Layout

Before the binary is reassembled, code layout is performed. The goal of goal layout is to place basic blocks in memory in an order that minimizes (a) the number of taken branches executed, (b) the number of instruction cache misses, and (c) the number of page faults incurred, and involves moving frequently executed blocks to one end of the address space and infrequently executed blocks to the other. The layout algorithm used in ILTO is described by Pettis and Hansen in [16]. Since basic blocks are moved during layout, branch instructions may be deleted, added, or may need to have their sense switched. If a block could be entered by means of a fall-through edge, then we have to insert an explicit branch if the block is moved. If we move a block so that its entry point immediately follows what had been a branch to the block, then we want to delete the branch to the block.

As a (somewhat artificial) example of code motion, consider the following C program fragment:

```
   if (x > 0 )
     { statements1; }
   else
     { statements2; }
```

Straightforward Itanium code for this would be

```
         cmp.gt p6,p7 = x,0 ;;
   (p7)  br.cond Else
         code for statements1
         br.cond Done
   Else: code for statements2
   Done:
```

If we decide to switch the positions of the code blocks for `statements1` and `statements2`, the only other change we need to make is to use `p6` to guard the predicate on the branch instruction. This is a safe transformation because `p6` and `p7` are strongly disjoint. This illustrates another use of predicate analysis.

5.4 Experimental Results

We evaluated our ideas using a set of seven programs from the SPECint-2000 benchmark suite: *bzip2*, *gzip*, *mcf*, *parser*, *twolf*, *vortex*, and *vpr*. The programs were run on an HP i2000 workstation with a 733 MHz Intel Itanium

[1]The compare instruction actually assigns the complement of `p6` to predicate register `p0`. However, since `p0` is hard-wired to the value *true*, the effect is to discard the complement.

Program	*Code Density*		S_1/S_0
	Original (S_0)	Optimized (S_1)	
bzip2	0.7011	0.7134	1.0175
gzip	0.7031	0.7127	1.0136
mcf	0.7012	0.7128	1.0165
parser	0.6985	0.7130	1.0208
twolf	0.6985	0.7121	1.0195
vortex	0.7300	0.7367	1.0091
vpr	0.6994	0.7134	1.0201
GEOMETRIC MEAN			1.017

(a) Code Density

Program	*Execution Time* (sec)		T_1/T_0
	Original (T_0)	Optimized (T_1)	
bzip2	1155.04	1002.59	0.868
gzip	1041.97	984.34	0.945
mcf	1506.34	1491.62	0.990
parser	1305.39	1266.66	0.970
twolf	1483.17	1405.97	0.948
vortex	1072.89	1001.57	0.934
vpr	1057.34	991.74	0.938
GEOMETRIC MEAN			0.941

(b) Execution time

Table 1: Performance: *gcc*-compiled programs

processor running Redhat Linux 7.1, kernel 2.4.3-12. The memory configuration of the system was as follows: split L1 instruction and data caches, each consisting of 16 KB of 4-way set associative cache memory with 32-byte lines; a 96 KB unified L2 cache; a 2 MB unified L3 cache; and 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.[2]

Static code density figures, expressing the ratio of useful (i.e., non-*nop*) instructions to the total number of instructions, were obtained as follows. For the input binaries, we measured code densities after first discarding unreachable code (in order to exclude code brought in by the linker from libraries that is not referenced by the program). Code densities after optimization were obtained just before the executables were written out and hence after all optimizations had been carried out. For these experiments, ILTO did not use any optimizations other than those described here, so the data presented reflect *only* the effects of if-conversion and predicate analysis.

Recall that, unlike August *et al.* [3], we postpone if-conversion until the end of the compilation process in order to keep our analyses and optimizations architecture-independent as far as possible. When evaluating our algorithm, therefore, there are two independent questions of interest: First, how effective is our algorithm at improving the performance of an unpredicated instruction stream, e.g., such as that produced by a conventional optimizing compiler that does not have specialized support for predication? Second, how effective is the algorithm in actually identifying available opportunities for if-conversion? The difference between the two is that it is possible, in principle, that we could obtain performance improvements from our if-conversion algorithm (the first question) even if it had weaknesses that caused it to miss a lot of optimization opportunities (the second question).

To address the first question, we evaluate our algorithm on programs compiled using the *gcc* compiler, which does not have very sophisticated facilities for dealing with predication; we used *gcc* version 2.96, at optimization level `-O3`.

[2]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.

Program	*Code Density*		S_1/S_0
	Original (S_0)	Optimized (S_1)	
bzip2	0.7023	0.7165	1.0203
gzip	0.7047	0.7191	1.0205
mcf	0.7010	0.7140	1.0186
parser	0.7042	0.7203	1.0229
twolf	0.7041	0.7200	1.0225
vortex	0.7220	0.7391	1.0236
vpr	0.7010	0.7150	1.0200
GEOMETRIC MEAN			1.021

(a) Code Density

Program	*Execution Time* (sec)		T_1/T_0
	Original (T_0)	Optimized (T_1)	
bzip2	843.65	820.16	0.972
gzip	633.15	648.86	1.025
mcf	1409.94	1419.79	1.007
parser	1190.45	1190.30	1.000
twolf	1267.49	1261.49	0.995
vortex	835.32	824.86	0.987
vpr	906.85	925.15	1.020
GEOMETRIC MEAN			1.001

(b) Execution time

Table 2: Performance: *ecc*-compiled programs

Table 1 gives performance results for this case. Table 1(a) shows code densities before and after optimization. It can be seen that our algorithm yields a slight improvement in code density of about 1.5%. Code density is improved by the if-conversion process, which replaces useless instructions, and by predicate analysis, which makes scheduling (and bundling) less constrained.

Table 1(b) shows the effect of our optimization on execution speed. The column labelled "Original" refers to the executable produced by *gcc*, while that labelled "Optimized" refers to the executable obtained using our if-conversion algorithm on the input binaries. The biggest speedup is obtained for the *bzip2* program, which improves by over 13%. On average, we see a speed improvement of 5.8%.

For the second question, we consider binaries obtained using Intel's *ecc* compiler version 5.0.1, at optimization level `-O3` together with profile feedback, i.e.: the programs were compiled with the options '`-O3 -prof_gen`,' then executed on the SPEC training inputs to generate profiles, and finally recompiled with the options '`-O3 -prof_use`,' Here we take input binaries that have already been heavily optimized by an industrial-strength, predicate-aware optimizing compiler using profile feedback; remove all predication using reverse if-conversion; then if-convert back using our algorithm. If there are significant weaknesses or imprecision in our algorithm, the quality of the code produced by our optimizer would be inferior to that of the input file, so we would see a performance degradation relative to the input binary. If, on the other hand, our approach is effective in identifying if-conversion opportunities, the performance of the code generated by ILTO should be comparable to that of the input binaries. Table 2 shows the performance numbers in this case. As shown in Table 2(a), our algorithm is actually able to improve static code densities by 2% on average compared to the original *ecc*-generated code. With respect to execution speed, as shown in Table 2(b), it can be seen that our algorithm produces code whose performance is essentially the same as that of the input *ecc*-optimized binaries. On three programs, *bzip2*, *vortex*, and *twolf*, our algorithm produces slightly faster binaries; on three others, *gzip*, *vpr*, and *mcf*, we get a slight slowdown. On average, the code obtained from ILTO is 0.1% slower than the original binaries. This indicates that in general, our predicate analysis and if-conversion algorithms are able to identify and recover pretty much all of the opportunities for if-conversion that were present in the input program but that were obfuscated during the initial reverse if-conversion phase.

6 Reverse Engineering Issues

As we described ILTO's organization in section 2, sandwiched between the front-end and the back-end of ILTO is the machine-independent optimization stage (which is not discussed in this paper). We have shown in the previous section that ILTO provides a solid foundation for implementing additional optimizations, since the algorithms used in the framework of ILTO do not have a large negative impact on performance, even when run on highly optimized code. However, because Itanium code is converted to an intermediate representation, the middle stage of ILTO may include any analyses and transformations, not just ones aimed at improving performance, without the those analyses and transformations needing to be predicate-sensitive. Therefore, we have also used ILTO as a platform for investigating reverse-engineering issues on the Itanium.

In a sense, unscheduling and unpredication, described in Section 4, can be classified as reverse-engineering transformations, since they result in code that is less tied to specific features of the Itanium (i.e. predication) and less mangled by compiler optimizations (i.e. scheduling), and is therefore easier to understand and analyze. But predication and scheduling are by no means the only contributors to program obfuscation. In this section, we describe how, using ILTO, we applied reverse-engineering techniques to speculation, another feature of the Itanium.

6.1 Speculation

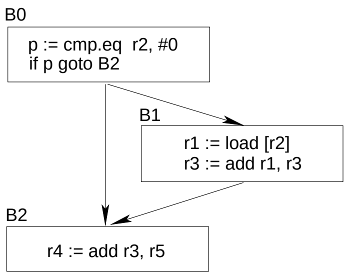
It is well known that processor speeds are growing faster than memory speeds, which means that the performance gap between the processors and memory is also growing steadily. One effect of this is that high-performance processors may be hamstrung because the memory system cannot deliver data as fast as the CPU would like. beyond what is possible using conventional instruction scheduling techniques, advanced architectures such as the Intel IA-64 (Itanium) have offered an innovative architectural feature: *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 [12]. However, speculation adds structure to generated code that does not reflect any logic in the original source, and 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.

In this section we present a technique for undoing low-level optimizations based on speculation in order to expose the original structure of speculative programs and thereby 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 model that supports the same speculative operations as the Itanium.

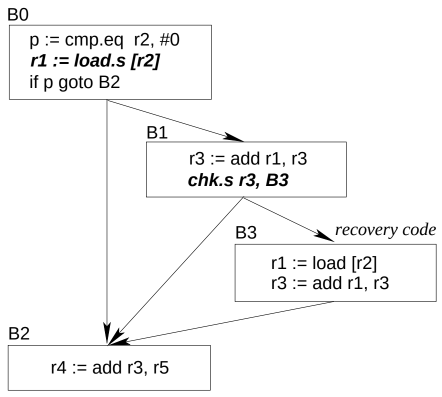
6.1.1 Background

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 9(a), where basic block B0 tests whether register r_2 contains a non-NULL value; 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 those of a normal load, but with one important difference: if the instruction generates an exception, such as segmentation or page fault, the exception is not handled immediately; 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



(a) Original unspeculated code



(b) Code with speculation

Figure 9: An example of control speculation

execution branches to recovery code provided by the compiler; otherwise, execution continues as normal. 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.

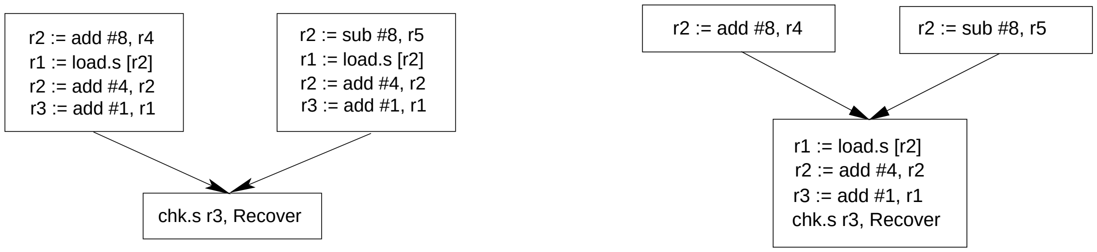
Using control speculation to the example shown above, we can move the load instruction above the preceding branch, in the process turning it into a speculative load. The resulting code, shown in

Figure 9(b), [3]

is considerably harder to understand than the original, for two reasons. 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, causing the speculative load to have 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.

6.2 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. Second, we verify that the speculative instruction can be safely replaced by an unspeculative load. Each of these steps is semantics-preserving.



```
r2 := add #8, r4
r1 := load.s [r2]
r2 := add #4, r2
r3 := add #1, r1
```

```
r2 := sub #8, r5
r1 := load.s [r2]
r2 := add #4, r2
r3 := add #1, r1
```

```
chk.s r3, Recover
```

(a) Before load sinking

```
r2 := add #8, r4
```

```
r2 := sub #8, r5
```

```
r1 := load.s [r2]
r2 := add #4, r2
r3 := add #1, r1
chk.s r3, Recover
```

(b) After load sinking

Figure 10: An example of load sinking

6.2.1 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, when a speculative load is encountered in a program, the very fact that a speculative load has been used—rather than an "ordinary" one—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 as part of unspeculation.

In this connection, 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 such loads, immediately before such a check instruction. 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, illustrated in Figure 10, as *load sinking*. Note that when a speculative load I is sunk, other instructions that depend from 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 $I \rightleftharpoons^\star J$.

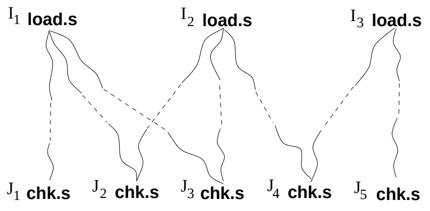


Figure 11: General structure of speculative computations

[3]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, where *op* denotes the operation, *dst* is the destination, and src_1, src_2, ... are the source operands:

dst := *op* src_1 src_2 ...

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* $[r]$.

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. This is illustrated in Figure 11. Moreover, we have to contend with the possibility that a speculation check may be associated with several different speculative loads, which may have different sets of dependent instructions associated with them. The remainder of this section addresses these issues in greater detail.

Finding relationships between instructions Our first goal is to identify, for a given speculative load, the set of associated check instructions that test whether that load succeeded or failed. As mentioned in Section 6.1.1, 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 is illustrated in Figure 9(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. Thus, 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` $r', \ldots$', say that J *checks* I if either of the following hold:

1. $r' \equiv r$, and the definition I of r reaches J;[4] or

2. there is an instruction I' that uses r and which 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 9(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.

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_1 and I_2, and the speculation check J_2, in Figure 11. Let S_1 be the set of instructions dependent on the speculative load I_1, and S_2 the set dependent on I_2. When sinking I_1 we want to move all the instructions in S_1 down to the check instruction; when sinking I_2, similarly, we want to move all of S_2. If $S_1 \neq S_2$ it is not clear what instructions ought to be moved down to the check; if this happens, load sinking is said to fail.

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

[4]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].

Definition 6.1 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 that 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 6.2 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, Figure 11 shows a total of eight distinct paths between the speculative loads and associated checks. Path independence requires that the instructions dependent on the speculative loads along each such pair of paths be the same.

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 12.

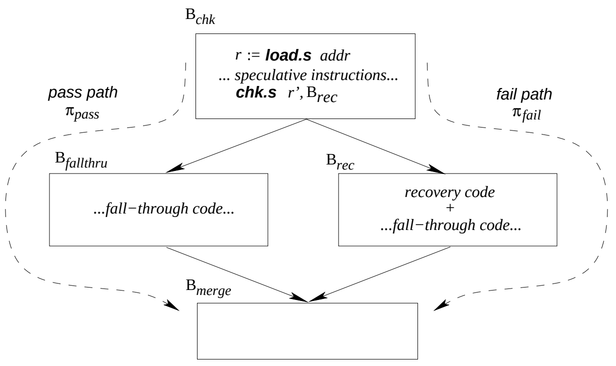


Figure 12: Code structure after load sinking

6.2.2 Recovery Code Verification

In the code resulting from sinking, shown in Figure 12, there are two possible outcomes for the speculation check in block B_{chk}. If the speculative load completes successfully without setting any NaT bits, execution takes the *pass path* $\pi_{pass} \equiv B_{chk} \rightarrow B_{fallthru} \rightarrow B_{merge}$. Or else the speculative load may fail and set NaT bits, in which case control goes through the recovery code along the *fail path* $\pi_{fail} \equiv B_{chk} \rightarrow B_{rec} \rightarrow B_{merge}$. 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:

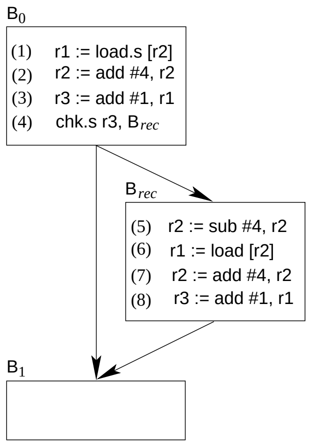


Figure 13: An Example of Recovery Code Verification

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 following example illustrates a situation where the load equivalence condition is not satisfied:

```
          r1 := NULL
          r2 := load.s [r1]
          chk.s r2, Rec

          r2 := 0    /* fall-through */
          br End

     Rec: r2 := 0    /* recovery */
          br End
```

It is easy to see that this code fragment satisfies the path equivalence criterion. In this case, the speculative load results in a deferred exception that causes the check to branch to the recovery code, where register r_2 is assigned the value 0. However, if we replace the speculative load by an unspeculative load, the result will be an exception that is not deferred, thereby changing the behavior of the program.

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 6.3 indicate, this suffices for most instances of speculation encountered in practice.

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. 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 10(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. This is illustrated in Figure 13 (which shows the speculation and recovery code corresponding to Figure 10(b)); the parenthetical numbers to the left of instructions are provided for convenience in referring to them. The first instruction in the recovery code sequence, 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.

To prove path equivalence for the code at a speculation check, we specify a set Φ describing values of the initial program state at the speculative load for which path equivalence does not hold—i.e., for some register or memory location x, the value of x along the pass path differs from its value along the fail path. We then attempt to show that $\Phi = \emptyset$ using constraint solving techniques. 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 formula A and a set of variables V, let

$$(\exists_{[V]})A$$

denote the formula where the existential quantification is over all of the variables in A except for those in V. Using this notation, we can write the specification of the set Φ as:

$$\Phi = \{\bar{x} \mid (\exists_{[\bar{x}]})[\Psi_p(\bar{x}) \wedge \Psi_f(\bar{x}) \wedge \Delta(\bar{x})]\}$$

where $\bar{x}$ is a tuple representing (the relevant portion of) the program state; $\Psi_p(\bar{x})$ and $\Psi_f(\bar{x})$ are formulae expressing the values of locations at the end of the pass path and the fail path, respectively, in terms of the initial values $\bar{x}$; and $\Delta(\bar{x})$ states 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.

In constructing these formulae, relationships between the values of registers can be expressed straightforwardly, but indirect memory accesses make it harder to reason about the contents of memory locations. Our current implementation is conservative in its treatment of memory: our treatment of memory aliasing is discussed in more detail in Section 6.2.3. This is not a significant problem in practice, however, since changes to memory via *store* instructions tend to be rare in recovery code. The discussion below focuses on reasoning about register values.

Assume that each instruction in the program is given a unique identifying number: the instruction with number k is written 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. There are two cases, depending on whether the analyzer knows the semantics of the $\oplus$ operation.

   If the semantics of $\oplus$ is known to the analyzer, then $C_k \equiv r_k^p = f_\oplus(s_i^p, t_j^p)$ where I_i is the most recent instruction that defines register s ($i = 0$ if s has not yet been defined along the pass path), I_j is the most recent instruction that defines register t ($j = 0$ if t has not yet been defined along the pass path), and $f_\oplus$ is a function that 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.

   If the semantics of the operation $\oplus$ is not known to the analyzer, then we cannot specify the set Φ, and path independence cannot be verified.

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 [17] to determine the satisfiability of the constraints defining the set Φ.

The algorithm can be illustrated using the recovery code shown in Figure 13. We have

$$\Phi = \{\bar{x} \mid (\exists_{[\bar{x}]})[\Psi_p(\bar{x}) \wedge \Psi_f(\bar{x}) \wedge \Delta(\bar{x})]\}.$$

where the tuples $\bar{x}$ are triples $\langle r1_0, r2_0, r3_0\rangle$. We have:

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

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 defines the empty set. This, in turn, implies path equivalence for the code in Figure 13.

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 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.

6.2.3 Memory Disambiguation

Memory disambiguation involves knowing 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. This is a difficult problem in general (e.g., see [11, 14]), exacerbated by the lack of semantic structure at the machine code level. Our current implementation generalizes a simple analysis technique known as *instruction inspection* [5, 20]. The general idea here is that two memory references can be inferred to be non-conflicting if either (i) they point to disjoint regions of memory, e.g., the stack and the global data area; or (ii) they use distinct offsets from the same base register r, with no intervening definitions of r.

Our implementation considers four mutually disjoint memory regions: the procedure stack, the heap, the static global data section, and the global offset table. The last of these deserves some explanation. In Itanium programs 64-bit constants (e.g., addresses) usually do not appear directly as immediate operands, but are loaded from a special region of memory called the global offset table.[5] To fetch an address from this table, an offset is added to a particular register (`r1`) designated as the global data pointer, which contains the address of the first entry in the table. There are two components to our analysis:

1. **Region Analysis.** We use a simple iterative dataflow analysis to associate, with memory reference in the program, a subset of these regions that the reference may access.
2. **Offset Computation.** For memory accesses that cannot be guaranteed to be in disjoint memory regions based on this analysis, we use a simple backward offset computation to determine whether they involve accesses at difference offsets from the same base address.

These are discussed in more detail in the following sections.

Region Analysis Region analysis is a dataflow analysis whose goal is to identify the memory region(s) that a register may point at. We start with the set of regions

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

where heap refers to heap locations, stack to stack locations, global to globals, GOT to the global offset table, and num to numerical constants. The analysis domain then is the powerset of this set, $\mathcal{P}(\mathsf{D})$, ordered by subset inclusion; $(\mathcal{P}(\mathsf{D}), \subseteq)$ forms a complete lattice, with least element $\emptyset$ denoting an unreachable reference, and greatest element $\mathbf{D}$ denoting an unknown value. Instructions within a basic block are handled as follows:

1. Register `r1`, and addresses computed by adding numerical constants to `r1`, point into the global offset table.
2. If a register r points into the global offset table, then an indirect load through r points to a global.
3. The stack pointer `sp`, and addresses computed by adding numerical constants to `sp`, point into the stack.
4. load($\{$ global, heap, stack $\}$) $\Rightarrow$ unknown
5. malloc(), alloc(), calloc() $\Rightarrow$ heap
6. location $\pm$ number $\Rightarrow$ location

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.

Offset Computation Given a memory access from register r_1 and another from register r_2, we can reason that these accesses do not overlap if the absolute value of the difference between r_1 and r_2, $|r_1 - r_2|$, is at least as large as the size of the memory being accessed (for instance, if both access four-byte words, then to guarantee that they do not overlap we must show that $|r_1 - r_2| \geq 4$). A simple way to determine the difference between two registers is to find another register r such that the values of r_1 and r_2 can be both be expressed as constant offsets off the value of r, i.e. $r_1 = r + c_1$ and $r_2 = r + c_2$, where c_1 and c_2 are known. Then the difference is simply $|c_1 - c_2|$. In order to find r we use the backwards computation shown in Figure 14.

[5]Other 64-bit architectures where the instruction width is smaller than 64 bits, e.g., the Compaq Alpha, use a similar approach for handling 64-bit constants.

Let I_1 and I_2 be two instructions in the same basic block B (such that I_2 is one or more instructions after I_1), which read from the memory locations stored in registers r_1 and r_2, respectively. Also, let w_1 and w_2 be the number of bytes of memory accessed by I_1 and I_2, respectively. The following algorithm returns TRUE if the two accesses might alias, and FALSE if they cannot.

```
let $r'_1 = r_1, r'_2 = r_2, c_1 = c_2 = 0$
/* Stage 1: analyze code backwards from $I_2$ to $I_1$ */
for each instruction $I$ from $I_2$ to $I_1$ do
  if $I$ is of the form $r'_1 = r + c$
    let $r'_1 := r$ and $c_1 := c_1 + c$
  else if $I$ writes to $r'_1$
    return TRUE; /* $r_1$ cannot be expressed as an offset */
  end if
end for

/* Stage 2: analyze code backwards from $I_1$ */
for each instruction $I$ from $I_1$ to the top of $B$ do
  if $I$ is of the form $r'_1 = r + c$
    let $r'_1 := r$ and $c_1 := c_1 + c$
  else if $I$ is of the form $r'_2 = r + c$
    let $r'_2 := r$ and $c_2 := c_2 + c$
  else if $I$ writes to $r'_1$ or $r'_2$
    return TRUE; /* The registers cannot be expressed as offsets */
  end if

  /* Check to see if the registers can now be expressed as offsets from the same register */
  if $r'_1 = r'_2$
    if $c_1 < c_2 \wedge |c_1 - c_2| \geq w_1$ return FALSE;
    else if $c1 > c_2 \wedge |c_1 - c_2| \geq w_2$ return FALSE;
    else return TRUE; /* The accesses overlap */
    end if
  end if
end for
```

Figure 14: The Offset Computation Algorithm

6.3 Experimental Results

To evaluate our ideas, we first created Itanium binaries that contained a large amount of speculated code. We compiled a set of benchmarks from the SPECint-2000 suite (*bzip2*, *gzip*, *mcf*, *parser*, *twolf*, *vortex*, and *vpr*) with Intel's *ecc* compiler version 5.0.1, at optimization level `-O3` together with profile feedback, i.e.: the programs were compiled with the options '`-O3 -prof_gen`,' then executed on the SPEC training inputs to generate profiles, and finally recompiled with the options '`-O3 -prof_use`.' This process produces binaries with a significant amount of control speculation.

The effectiveness of our unspeculation algorithm can be measured both 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 number of speculative loads and speculation checks in the program before unspeculation to the number after unspeculation. In general, the more often our algorithm can untangle a

Program	SPECULATIVE LOADS			SPECULATION CHECKS		
	Orig. (L_0)	Unspec. (L_1)	Improvement (%) ($(L_0 - L_1)/L_0$)	Orig. (C_0)	Unspec. (C_1)	Improvement (%) ($(C_0 - C_1)/C_0$)
bzip2	130	31	0.762	124	42	0.661
gzip	224	62	0.723	181	54	0.702
mcf	94	31	0.670	97	34	0.649
parser	483	85	0.824	451	75	0.834
twolf	1542	385	0.750	1399	354	0.747
vortex	5339	451	0.916	5217	352	0.933
vpr	608	152	0.750	614	145	0.764
GEOM. MEAN:			0.767			0.750

Table 3: Amount of speculated code before and after unspeculation

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 4: Effects of unspeculation on program size

speculative region, the higher this ratio will be. Table 3 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 by about 79.5% and the number of speculation checks by about 78.2% 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 4. This table shows that, on average, the number of instructions decreased by about 6.8%, the number of basic blocks decreased by about 14%, and the number of edges decreased by about 12.7% 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 a compiler optimization, we expect that unspeculation results in less efficient code. To test this, we ran the same timings tests as described in Section 5.4 on the original binaries and on the binaries after unspeculation. The results of these tests are shown in Table 5. This table shows that the unspeculated binaries suffer a performance hit of about 6% on average.

7 Related Work

If-conversion has been investigated by Mahlke *et al.*, who discuss the formation and use of hyperblocks—single entry multiple-exit collections of basic blocks [13]. The focus of their work, by contrast with that described here, is in

Program	*Execution Time* (sec)		T_1/T_0
	Original (T_0)	Unspeculated (T_1)	
bzip2	843.65	843.17	0.999
gzip	633.15	675.51	1.067
mcf	1409.94	1434.44	1.017
parser	1190.45	1227.04	1.031
twolf	1267.49	1336.75	1.055
vortex	835.32	1009.82	1.209
vpr	906.85	969.03	1.069
GEOMETRIC MEAN			1.062

Table 5: Performance

identifying which set of blocks should be included in a hyperblock. Once a hyperblock has been formed, if-conversion is used to transform it into a single basic block containing predicated instructions, which is very different from what we do. August *et al.* discuss the tradeoffs associated with the timing of if-conversion in the overall compilation process [3]. They advocate an approach dual to ours, namely, carrying out aggressive if-conversion early in the compilation process, using compiler analyses and optimizations that understand predicated code, and then selectively reverse-if-convert during scheduling where appropriate. We have shown that it is possible to get excellent performance without requiring analysis and optimization phases to understand predicated code.

Mahlke *et al.* use the notion of *predicate hierarchy graphs* to keep track of relationships between predicates [13]. Their analysis is based on keeping track of which predicates guard the definition of other predicates, and so does not work well when predicate relationships are not hierarchical. Eichenberger and Davis describe an analysis that collects logical expressions expressing relationships between predicates [6]. A more precise approach, based on keeping track of logical partitions between predicate expressions, is described by Gillies et al. [8] and Johnson and Schlansker [10]. None of these analyses extend across join blocks, i.e., where multiple control flow paths merge. Sias, Hwu and August discuss the efficient implementation of predicate analyses using binary decision diagrams, and extend prior work to handle general control flow [19]. The analysis described here, by contrast, takes a very different approach. It is formulated within the framework of a traditional meet-over-all-paths dataflow analysis, which makes it relatively straightforward to understand, implement, and extend in various ways, e.g., to inter-procedural analysis. We have already extended our analysis to a context-insensitive inter-procedural predicate disjointness analysis, and we are currently investigating the question of context-sensitive inter-procedural disjointness analysis.

8 Conclusions and Future Work

In this paper we have presented new approaches for optimizing, analyzing, and reverse-engineering Itanium code, and have described a system, ILTO, that implements our ideas. We have shown that ILTO— which removes traces of machine-dependent optimizations such as predication from the input binary in order to render the code more amenable to traditional analysis, and which delays if-conversion until the tail-end of the rewriting process—can nonetheless generate efficient code (about 6% faster than code generated by *gcc* over the SPECint-2000 benchmark suite). Therefore the organization of and algorithms used by ILTO provide a good starting point for implementing further optimizations. In addition, we used ILTO as a platform for investigating unspeculation, and developed algorithms that effectively perform this reverse-engineering transformation.

Much work remains to be done on the core optimization stage of ILTO. Though ILTO provides an intermediate form of Itanium code free of predication, some existing optimization algorithms must need be tweaked to deal with the particular instruction and register set of the Itanium. Others are more difficult to port: for instance, function inlining is complicated by the existence of register windows on the Itanium, and implementing this optimization would involve complicated analysis. Finally, there are Itanium-specific optimizations that would be interesting to evaluate, such as optimizations involving the global offset table, which heavily used by the Itanium but absent in other architectures. On the other hand, it would also be interesting to use ILTO as a framework for investigating other Itanium-specific "unoptimizations," such as converting software-pipelined loops to conventional loops.

9 Acknowledgements

I would like to thank my advisors, Dr. Greg Andrews and Dr. Saumya Debray, without whom the work presented in this paper could not have been possible. This work was also supported by the National Science Foundation under grants CCR-0073394, EIA-0080123, and CCR-0113633.

References

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

[2] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proc. Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 177–189, January 1983.

[3] D. I. August, W. W. Hwu, and S. A. Mahlke. A framework for balancing control flow and predication. In *Proc. 30th Annual International Symposium on Microarchitecture*, pages 92–103, 1997.

[4] Y. Choi, A. Knies, L. Gerke, and T.-F. Ngai. The impact of if-conversion and branch prediction on program execution on the Intel Itanium processor. In *Proc. 34th Annual International Symposium on Microarchitecture*, pages 182–191, December 2001.

[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] A. E. Eichenberger and E. S. Davidson. Register allocation for predicated code. In *Proc. 28th Annual International Symposium on Microarchitecture*, pages 180–191, 1995.

[7] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proc. ACM SIGPLAN 86 Symposium on Compiler Construction*, pages 11–16. June 1986.

[8] 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.

[9] S. Haga and R. Barua. EPIC Instruction Scheduling Based on Optimal Approaches. In *Proc. First Annual Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Technology*, 2001.

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

[11] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

[12] 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.

[13] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *25th Annual International Symposium on Microarchitecture (MICRO-25)*, pages 45–54, 1992.

[14] R. Muth and S. K. Debray. On the complexity of flow-sensitive dataflow analyses. In *Proc. 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-00)*, pages 67–80, January 2000.

[15] E. W. Myers, Jr. A precise inter-procedural data flow algorithm. In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages (POPL '81)*, pages 219–230, January 1981.

[16] K. Pettis and R. C. Hansen. Profile-guided code positioning. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 16–27, June 1990.

[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. Plto: A link-time optimizer for the Intel IA-32 architecture. In *Proc. 2001 Workshop on Binary Translation (WBT-2001)*, 2001.

[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] D. W. Wall. Speculative execution and instruction-level parallelism. Technical Report TN-42, Digital Equipment Corporation, Western Research Lab, March 1994.