Predicate Analysis and If-Conversion in an Itanium Link-Time Optimizer *

Noah Snavely, Saumya Debray, Gregory Andrews

Department of Computer Science
University of Arizona
Tucson, AZ 85721.

{snavely, debray, greg}@cs.arizona.edu

ABSTRACT

EPIC architectures, such as the Intel IA-64 (Itanium), combine explicit instruction-level parallelism with instruction predication. To generate efficient code, it is important to use predication effectively. In particular, it is important to replace conditional branches and multiple code blocks by single, branch-free code blocks when doing so would lead to faster code. This process, which is known as if-conversion, is generally carried out early in the code-generation process; hence subsequent analyses and optimizations have to deal with predicated code. This paper examines an alternative approach in which code is unpredicated during disassembly, the internal representations are virtually identical to those in a conventional architecture (specifically the IA-32 Pentium) and if-conversion is done late in the compilation process, at the same time as instruction scheduling and just before code layout. This paper also presents new algorithms for analyzing predicated code and evaluates their efficacy. We show that our approach is able to produce code that is denser (fewer nop instructions) and almost as fast as the best code produced by the Intel `ecc` compiler on the SPECint-2000 benchmark suite. On the same programs, our predicate analysis and if-conversion algorithms lead to an average speed improvement of a little over 4% on the best code produced by the `gcc` compiler.

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 support predicated instructions and explicit instruction-level parallelism. 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.

In order to make effective use of the capabilities of such architectures, a compiler must selectively eliminate conditional branches 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. There are two important questions that have to be addressed in this regard. The first is that of when if-conversion should be carried out in the compilation process. The second is that of determining relationships between predicates, which is necessary to identify dependencies between predicated instructions.

One option for carrying out if-conversion is to do it 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 of such an approach is that analyses and optimizations in the compiler backend may have to be reimplemented to cope with predication. An alternative is to delay if-conversion until much later in the compilation process, during instruction scheduling, after most optimizations have been carried out. The advantage here is that other analyses and optimizations do not have to be made predication-aware. This can simplify the construction of portable multi-target optimizers.

The determination of relationships between predicates can be helpful for improving the quality of code generated. For example, during instruction scheduling, it can allow the compiler to exclude false scheduling dependencies between instructions whose guarding predicates cannot be simultaneously true. Another situation where knowledge of such relationships can be useful is in the context of profile-guided code layout, which can enhance program performance by improving its instruction cache behavior [12]. This can sometimes require us to invert the sense of a branch, e.g., so as to have it fall through rather than be taken. If the branch instruction in question is conditioned on a predicate register p, and we know that a predicate q is guaranteed to be the complement of p at that point, we can achieve this inversion simply by replacing the guard predicate p by its complement q. Without such knowledge, we may have to explicitly compute the complement of p, which can take several instructions and adversely affect performance.

The contribution of this paper is to present algorithms for if-conversion and predicate analysis and to evaluate their efficacy experimentally. Our approach is very different from those that have been discussed in the literature. To simplify implementation and keep our analyses and optimizations architecture-independent, our internal representation does not support predicated instructions. We therefore carry out reverse if-conversion after disassembly, converting instructions into their unpredicated form and introducing conditional branches in the process, following which a conventional control flow graph is constructed. This control flow graph is then subjected to various analyses and optimizations in the usual way. We then carry out if-conversion during instruction scheduling, just prior to code layout. Our approach can be seen as dual to that of August *et al.* [3]. Our predicate analysis is also quite different from previous proposals for determining relationships between

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

predicates: we use a dataflow analysis that is able to handle arbitrary control flow, and can be extended in a straightforward way to an inter-procedural analysis.

The remainder of the paper is organized as follows. Section 2 gives background information on the Itanium architecture and on our experimental optimization system. Section 3 describes how we analyze the use of predicate registers to compute what we call weak and strong disjointness sets. Section 4 presents our if-conversion algorithm. Section 5 contains examples that illustrate the use of predicate analysis and if-conversion. Section 6 address two questions: How effective is our approach in improving a mostly unpredicated instruction stream? And how effective is it in identifying opportunities for if-conversion? That section describes our experimental method and shows that the answers to both questions are positive. 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. Then we give an overview of the processing stages in ILTO and identify the places where it employs predicate register analysis.

2.1 Itanium Architecture

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.

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.

2.2 ILTO: 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 [13]. 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.

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 *inter-procedural control flow graph* (e.g., see [11]). 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 1, 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 modelled 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,

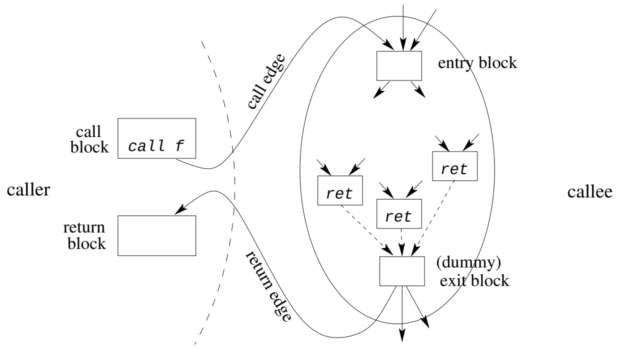


Figure 1: Representing function calls in the interprocedural control flow graph

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 ILTO system has nine major stages: The ILTO system has nine major stages:

1. *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.

2. *Predicate Analysis.* Compute predicate register disjointness sets, as described in Section 3.

3. *Unpredicate the CFG.* Remove predication from the instructions in the CFG by constructing explicit decision nodes.

4. *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.

5. *Predicate Analysis.* Recompute predicate register disjointness sets.

6. *Scheduling and If-Conversion.* Form a schedule for each basic block and convert decision nodes to predicated instructions where possible. Group instructions into bundles.

7. *Predicate Analysis.* Recompute predicate register disjointness sets.

8. *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.)

9. *Global Bundle Check and Patch.* Iterate through the basic blocks to check the validity of instruction bundles and to repair them when needed.

Predicate disjointness sets specify relations between the values of predicate registers at the start of each basic block. They are consulted as the CFG is unpredicated in order to simplify some cases, used extensively during if-conversion and instruction scheduling to produce good code, and used when blocks are moved during code layout in order to change the sense of branches. The definition, construction, and use of predicate disjointness sets are described in the next section.

3. PREDICATE ANALYSIS

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:

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* if $p \Leftrightarrow \neg q$. $\blacksquare$

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.

An 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.

In order to do effective 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, and which other registers are weakly disjoint from it? (There can be at most one register that is strongly disjoint, but there could be several that are weakly disjoint.) 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 two kinds of such sets at each basic block B:

```
initialize weakIN(B) and strongIN(B) as described in the text;
weakOUT(B) = weakIN(B) ; strongOUT(B) = strongIN(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 weakOUT(B) and strongOUT(B) ;
          add (pA, pB) to weakOUT(B) and strongOUT(B) ;
      else
          set wasInWeak to true if (pA, pB) is in weakOUT(B) and to false otherwise;
          set wasInStrong to true if (pA, pB) is in strongOUT(B) and to false otherwise;
          remove all pairs containing pA or pB from weakOUT(B) and strongOUT(B) ;
          if wasInStrong then
              add (pA,pB) to weakOUT(B) and strongOUT(B) ;
          else if wasInWeak or pG == pA or pG == pB then
              add (pA,pB) to weakOUT(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 weakOUT(B) and strongOUT(B) ;
      add (pA, pB) to weakOUT(B) ;
      for all (p, pG) that are in weakOUT(B) do
          add (p, pA) and (p, pB) to weakOUT(B) ;
      end for
  else /* I is a parallel AND or OR compare instruction */
      remove all pairs containing pA or pB from weakOUT(B) and strongOUT(B) ;
  end if
end for
```

Figure 2: Computing Predicate Disjointness Sets for a Basic Block

DEFINITION 3.2. 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. ▮

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

1. 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.

2. Determining disjointness 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:

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

   (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 disjointness 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:

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

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

   $$weak\mathsf{IN}(B) = \bigcap_{P \in preds(B)} weak\mathsf{OUT}(P), \text{ and}$$
   $$strong\mathsf{IN}(B) = \bigcap_{P \in preds(B)} strong\mathsf{OUT}(P).$$

Figure 2 gives the algorithm for computing $weak\mathsf{OUT}(B)$ and $strong\mathsf{OUT}(B)$ from $weak\mathsf{IN}(B)$ and $strong\mathsf{IN}(B)$. 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

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

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 disjointness relations at the call site. A simple conservative estimate for intra-procedural analyses is to assume that nothing is known about disjointness 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 disjointness 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. 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 3 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:

1. We attempt to replace nops 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 a 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 3 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 pred-

[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.

```
for each basic block B in the function do
  1. schedule B;
  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 3: The Basic If-Conversion Algorithm

icate 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. EXAMPLES

As described in Section 2.2, we analyze predicates three times in ILTO: before unpredicating the control flow graph, before if conversion and scheduling, and before code layout. Below illustrate how disjointness sets are used to simplify control flow graphs, to produce compact code during if conversion, and to reverse the sense of branches during code layout.

5.1 Unpredicating the Control Flow Graph

When ILTO disassembles an Itanium binary, it first unbundles instructions, determines basic blocks, and constructs a control flow graph (CFG); at this point, instructions in basic blocks are still predicated. We then unpredicate the instructions, replacing guard predicates by decision nodes and adding new basic blocks and edges to the CFG. Often we can simplify the structure of the unpredicated CFG by taking account of the semantics of predicate instructions. Having a less-complicated CFG simplifies later analyses and makes it easier to produce efficient code later on.

Often the source program contains code sequences that have the following structure:

```
      cmp.eq p6,p7=r10,r11 ;;
(p6)  instr1
(p6)  instr2
(p7)  instr3
```

This kind of machine code results from source code having the form (in pseudo-C):

```
if (condition)
  { instr1; instr2; }
else
  { instr3; }
```

The machine code uses if-conversion and predication to avoid two branches: one to jump to the else block and one to jump over the else block (from the end of the then block).

The straightforward way to unpredicate the above machine code would be to create two decision nodes—one to test p6 and one to test p7—and and two code blocks, as shown in Figure 4(a). However, the compare instruction makes p6 and p7 strongly disjoint, and they remain strongly disjoint while the instructions are executed. Hence, we can create a simpler control flow graph in which (a) there is a single conditional branch at the end of a basic block B0, (b) that block has a true edge to a block B1 containing the instructions that were predicated on p6 and a false edge to a basic block B2 containing the instructions that were predicated on p7. In short, we get the simpler, diamond-shaped control flow graph shown in Figure 4(b).

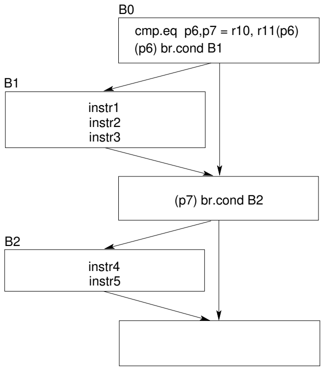
5.2 Producing Compact Code

The following example shows how weak disjointness sets are used during if-conversion to produce compact, efficient code.

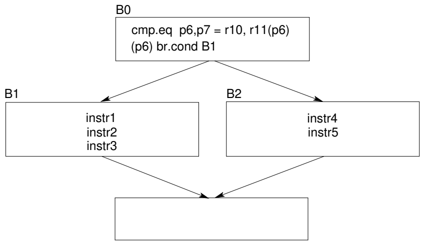
Consider the following C code fragment:

```
if (x == 0) {
  if (y == 0) z = 0; else z = 1;
}
else
  z = 2;
```

A straightforward translation of this to Itanium code would have the following structure:



(a) Naive unpredication



(b) Unpredication using disjointness information

Figure 4: An example of unpredication using predicate analysis

```
            cmp.eq p6,p7=x,0 ;;
      (p7)  br.cond L2
            cmp.eq p8,p9=y,0 ;;
      (p9)  br.cond L1
            mov z=0
            br.few Done
L1:         mov z=1
            br.few Done
L2:         mov z=2
Done:
```

This is the traditional way of handling conditionals. However, we can collapse the inner if/then/else statement—from the second comparison above through the last branch—into the compare, two predicated moves, and the last branch, as follows:

```
      cmp.eq p6,p7=x,0 ;;
(p7)  br.cond L2
      cmp.eq p8,p9=y,0 ;;
(p8)  mov z=0
(p9)  mov z=1
      br.few Done
L2:   mov z=2
Done:
```

This is called if-conversion; it depends on recognizing that p8 and p9 are strongly disjoint and hence that only one of the two moves will actually be executed.

We can if fact do even better for this type of code sequence: compact the code into a single basic block with *no* branches. After the first compare, p6 and p7 are strongly disjoint. The first branch and the instruction at L2 are executed if p7 is true. If p6 is true (and hence x==0), then the second compare and one of the move instructions predicated on (p8) or (p9) will be executed. In short, using predicate analysis we can determine that the three moves are mutually independent, and we can simplify the code to a single block as follows:

```
      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
```

In fact, the three moves can even be scheduled in the same instruction group and hence execute in parallel. The second instruction uses an unconditional compare so that both p8 and p9 are cleared before the compare, and hence they are false if p6 is false. This kind of code appears quite frequently in binaries produced by Intel's *ecc* compiler. ILTO is able to produce it by using predicate analysis, which leads to the following three inference chains:

p8 ⇒ p6 ⇒ ¬ p7	p7 and p8 weakly disjoint
p9 ⇒ p6 ⇒ ¬ p7	p7 and p9 weakly disjoint
p8 ⇒ ¬ p9	p8 and p9 weakly disjoint

5.3 Branch Sense Reversal During Code Layout

The final example arises during code layout, which places basic blocks in memory in an order that attempts to minimize the number of instruction cache misses. This involves moving frequently executed blocks to one end of the address space and infrequently executed blocks to the other. 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.

6. 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 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`. 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 1.3%. 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 12%. On average, we see a speed improvement of 4.3%.

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 a good, 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.

Program	*Code Density*		S_1/S_0
	Original (S_0)	Optimized (S_1)	
bzip2	0.7011	0.7119	1.0154
gzip	0.7031	0.7117	1.0123
mcf	0.7012	0.7098	1.0122
parser	0.6985	0.7088	1.0147
twolf	0.6985	0.7103	1.0168
vortex	0.7300	0.7338	1.0051
vpr	0.6994	0.7117	1.0177
GEOMETRIC MEAN			1.013

(a) Code Density

Program	*Execution Time* (sec)		T_1/T_0
	Original (T_0)	Optimized (T_1)	
bzip2	1155.04	1015.06	0.879
gzip	1041.97	987.82	0.948
mcf	1506.34	1500.58	0.996
parser	1305.39	1269.69	0.973
twolf	1483.17	1458.61	0.983
vortex	1072.89	1042.48	0.972
vpr	1057.34	1008.77	0.954
GEOMETRIC MEAN			0.957

(b) Execution time

Table 1: Performance: *gcc*-compiled programs

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 [10]. The focus of their work, by contrast with that described here, is in 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

[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

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 [10]. 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 [5]. A more precise approach, based on keeping track of logical partitions between predicate expressions, is described by Gillies et al. [7] and Johnson and Schlansker [9]. 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 [14]. 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.

For instruction scheduling we use a conventional list scheduling algorithm [6]. Our instruction bundling algorithm is similar to one in [8], but we augmented it to handle several special cases.

8. CONCLUSIONS

This paper has examined a new approach to dealing with predication in an EPIC architecture and presented new algorithms for predicate analysis and if-conversion. We converted a link-time optimizer (PLTO) for a conventional architecture, which did not support predication or explicit instruction-level parallelism, into one (ILTO) for an EPIC architecture, the IA-64 (Itanium), focusing on getting maximum mileage with minimal disruption. In particular, we wanted to leave the code analysis and optimization phases alone as much as possible, which meant that they would not be aware of predication or instruction-level parallelism.

The ILTO system deals with predication in three places: when unpredicating the control flow graph, when doing if-conversion and scheduling, and during code layout. ILTO deals with ILP only when scheduling and bundling instructions. We have developed the notion of predicate disjointness sets to guide these processes. Our predicate analysis is used during unpredication to produce simpler control flow graphs (which also turn out to be easier to get good code from); heavily during if-conversion to eliminate branches, increase ILP, and increase code density; and during code layout to changes the sense of branch instructions.

The results in Section 6 show two things. First, when given code that does not have very sophisticated use of predication (i.e., code from *gcc*), ILTO produces code that is on average over 4% faster and 1.3% denser on the SPECint-2000 benchmark suite. When given code that makes sophisticated use of the Itanium's features (i.e., code from *ecc*), ILTO produces code that is on average 2% denser and only 0.1% slower on the SPECint-2000 suite. In both cases, ILTO used only predicate analysis and if-conversion to improve the code; we did not examine other optimizations such as constant propagation or inlining. We are currently integrating these (mostly) architecture independent optimizations into ILTO.

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

[6] 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.

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

[8] 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.

[9] R. Johnson and M. Schlansker. Analysis techniques for

predicated code. In *Proc. 29th Annual International Symposium on Microarchitecture*, pages 100–113, 1996.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.