

# Phi-Predication for Light-Weight If-Conversion

Weihaw Chuang Brad Calder Jeanne Ferrante

Department of Computer Science and Engineering  
University of California, San Diego  
{wchuang,calder,ferrante}@cs.ucsd.edu

## Abstract

*Predicated execution can eliminate hard to predict branches and help to enable instruction level parallelism. Many current predication variants exist where the result update is conditional based upon the outcome of the guarding predicate. However, conditional writing of a register creates a naming problem for an out-of-order processor, and can stall the issuing of instructions. This problem arises from potential multiple predicated definitions reaching a use, which is unresolved until the prior predicate values are computed.*

*In this paper we focus on a light-weight form of predication, Phi-Predication, where all predicated instructions write a result value to their register regardless of the predicate value (i.e. even if it is false). Therefore, the predicate does not guard the writing of the result register; it instead acts as a form of selection between two input registers. This eliminates the naming problem for an out-of-order processor. Our Phi-Predicated ISA is derived from the predicated features of the Multiflow ISA, with extensions to efficiently predicate complex control flow. Our compiler modifications also expand upon prior techniques to provide efficient code generation. We examine the use of Phi-Predication for an in-order and out-of-order architecture and compare its performance to using select-op and IA64 ISA predication.*

## 1 Introduction

In today's deep pipeline architectures, the branch prediction penalty is significant. Using predication to perform if-conversion can remove mispredicted branches. By eliminating branches, if-conversion also encourages greater instruction fetch throughput, since fewer branches need to be predicted per cycle.

Several predicated ISAs have been proposed ranging from fully predicated ISAs, where almost every instruction can be guarded by a predicate, to very light-weight predication provided by conditional-move or select operations. Table 1 provides a high level breakdown of four previously proposed predicated ISAs, and the Phi-predication we propose in this paper. The columns list (1) the instruction types that can be guarded by a predicate, (2) how the predicates are

used to update the register definition for a predicated instruction, and (3) a coarse rating of the complexity of implementing the form of predication.

An important classification for a predicated ISA is how it performs its qualified register write. *Conditional-Writer* predicated ISAs guard the writing of an instruction's register definition by a predicate register. If the predicate is true, the destination register gets assigned a new value, otherwise it retains the old value. The IA64 and PlayDoh predicated ISA falls under this category [10, 12], as does the conditional move (CMOV) operations like the Alpha ISA [22]. In comparison to these, a *select* predicated ISA always writes the predicated definition by selecting from different input operands or constants. The Multiflow ISA provided a *SELECT* operation that used a general purpose register to select one of two values to update the select's definition register.

Conditional writer predication creates complexity for an out-of-order hardware scheduler through an effect called the *multiple-definition* problem. If there are multiple definitions of the same register guarded on different predicates, and the predicates have not been resolved, then it is unknown which definition the *use* should be linked to. Figure 1 illustrates the problem. The predicate generating CMPEQ instruction has yet to reach the execute pipeline stage, thus the predicate result is unavailable to the predicates of the MOV instructions. Consequently the ADD does not know which MOV is the correct assignment to R33. This creates an issue for an out-of-order processor, since it aggressively wants to speculatively schedule instructions, but is forced to delay the scheduling of the ADD until it knows which MOV the definition should be forwarded from.

Although this renaming ambiguity is not an issue for an in-order processor, our work also eliminates the additional complexity of a predicate aware full-bypass network. In this manifestation of the multiple definition problem, the bypass has to determine the last writer which potentially depends on some qualifying predicate, adding complexity and propagation delay. This issue was mentioned in [21], and alluded to in [19].

In this paper we examine a light-weight form of select predication, called *Phi-Predication*, where only a small set of

ISA	Type of Predicated Insts	Register Def	Complexity
IA64 and PlayDoh	Almost All Instructions	Conditional Write	High
Cydra 5	All Multi-Op	Conditional Write	Medium
Multiflow 500	Select/Stores/Floating-Point	Select & Cond-write	Low
Alpha	CMOVs	Conditional Write	Low
Phi-predication	Phi/Loads/Stores/Predicate Defs	Always Write	Low

Table 1: Predicated ISA Breakdown.

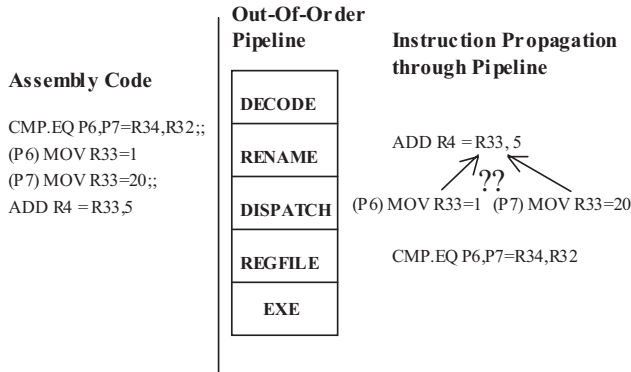


Figure 1: Multiple Definition Problem: Which value does R33 take at for its use at the ADD instruction? This ambiguity causes performance issues for out-of-order machines with predication.

the instruction types are predicated, yet we are still able to efficiently perform if-conversion on complex control flow. Our predicate ISA is derived from the predicated mechanisms used in the Multiflow architecture, which used a form of select predication [14, 5]. It is called Phi-predication because we essentially use the same insertion location as  $\phi$ -functions found by Static-Single Assignment (SSA) [6].

The overriding principle for the design of our Phi-predication ISA was to make register writing instructions *always-write* a value to the destination register when executed. Therefore, there can be only one potential value at a given time in the region for a given register use. The benefit of resolving the multiple definition problem in software with select predication for an out-of-order processor is that it requires substantially less hardware, yet provides good predication support. The downside is that it can lengthen the data dependency chain for certain regions. One of the goals of our compiler is to minimize this problem.

To process code regions larger than simple hammocks we need to qualify memory and control flow state updates yet obey the register always-write constraint. Our Phi-predication ISA provides additional instructions to do this. For memory operations, we provide predicated forms of load and store instructions for aggressive speculation. Any memory updates and exception handling are qualified by the predicate. A qualified false load will still update its definition register, writing a dummy (zero) value. For updates to predicate

registers, we borrow the unconditional compare instruction from IA64 [10] for our ISA, which always updates the predicates even if the compare is qualified false.

Another benefit of Phi-predication and other forms of light-weight predication is their lower opcode footprint, implementation cost, and complexity. In comparison, heavy-weight predicated ISAs like IA64 [10] predicate nearly all update instructions including ALU operations, memory, branches, and many special register state update. They also require a complicated predicated full-bypass or renaming machinery to correctly resolve the multiple definition problem.

We implement Phi-predication in the Intel Electron compiler and support its instructions in our IA64 version of SimpleScalar [2]. This allows us to provide a performance comparison of Phi-predication to the fully predicated IA64 ISA used in the Electron compiler. We compare the performance of Phi-predication to IA64 predication for in-order and out-of-order processors. This comparison is also done for an architecture with minimal predication support in the form of a SELECT (similar to CMOV) instruction.

The rest of the paper is organized as follows. Section 2 describes related work. Section 3 illustrates the instruction set, and introduces the compiler transformation. In section 4, we describe Phi-predication code generation. Section 5 provides the methodology for the experiments, and section 6 describes our performance results and our Phi-predication characterization.

## 2 Related Work

In this section we describe and contrast prior predicated ISAs, and describe prior work towards solving the multiple definition problem for an out-of-order IA64 processor.

### 2.1 Early Predicated ISA

Some of the earliest forms of predication are found in the Multiflow and Cydra machines. These machines focused on fast in-order, high ILP execution, as compared to our main interest in out-of-order execution. The Multiflow 200 and 300 series [14] implements only a select operation, while the 500 series [5] also implements conditional store and floating-point instructions. A select operation instruction takes two data inputs registers, an input selector operand that chooses between the two, and always-writes to an output destination

register. We later simulate and compare Select-op predication to Phi-predication. In this Select-op implementation we only use the three operand select and unconditional compare instructions, whereas Phi-predication has additional predicated memory operations and predicate OR operations.

An early form of heavy-weight conditional-writer predication architecture is found in the Cydra 5 [8, 17]. It is used to support if-conversion and software-pipelining. Every wide-word operation (MultiOp) can be made conditional on a bit in the Iteration Control Register, which is basically a predicate register file.

Many RISC and some CISC architectures adopted a CMOV instruction to handle simple if-conversion where the destination register is conditionally updated. This was exemplified by the Alpha ISA [22]. When Alpha moved to an out-of-order execution model, it ran into the multiple definition problem. This was solved by converting the CMOV into a two micro-op select where the destination register becomes the 2nd source register as described by Kessler [13].

## 2.2 Recent Light-Weight Predicated Research

Jacome et.al. [11] developed an SSA-based algorithm for if-conversion called Predicated Switching, targeted at clustered microarchitecture. At  $\phi$ -functions they generate conditional moves that sometimes can combine with regular instructions to become conditional-write predicated instructions. In comparison, we focus on a purely light-weight write-always approach.

Mahlke et.al. [15] provided a comparison of conditional-writer predication (full) and CMOV based predication (partial). Their partial predication and our Phi-predication are both light-weight implementations. Our ISA differs in the following: Our memory instructions qualify exceptions, while theirs are treated speculatively. We provide unconditional compares, and predicate OR instructions to reduce the predicate scheduling height between predicate generation to use. They use a more traditional RISC approach to boolean evaluation. The treatment for other exception causing instructions (floating point) also differs. Our compiler infrastructure is distinct in that we use an SSA framework to directly generate Phi-predication. They instead transform regions to full predication, then eliminate predicated instructions through predicate-promotion and peephole optimize away extra CMOV's. While their examples show efficient code generation, the compile time cost maybe large.

## 2.3 PlayDoh and IA64

After Cydra-5 a series of architectures were created supporting heavy-weight conditional-writer predication. HP labs developed PlayDoh as described in [12, 18] as a testbed for new VLIW techniques. Subsequently HP and Intel collaborated to implement an evolved VLIW architecture called IA64 [10, 19]. It features a heavy-weight conditional writer predication model for an in-order VLIW architecture. Virtually all non-system instructions can be predicated. We differ

from Cydra 5, PlayDoh and IA64 in that we use a select form of predication for all register and predicate defining instructions, with far fewer instructions specialized for predication, and overall lower implementation cost.

## 2.4 Prior Hardware Solution for Multiple Definition Problem

Wang et.al. [24] recognized that multiple definitions would be a problem in the renaming stage of an out-of-order implementation of IA64. The renaming stage is used to give each definition of an architectural register a unique physical name (removing WAW and WAR dependencies). With conditional-writer predication, it is possible to have multiple instructions guarded by different predicate registers write to the same architectural register. When a use of this architectural register is encountered in the rename stage, the values of the predicates may be required to determine which physical register to map to the architectural register. If the predicate values are not yet available, a stall must occur.

In an effort to remove as many unnecessary stalls as possible, Wang et. al. proposed the use of the select- $\mu$ op instruction in hardware for an IA64 out-of-order execution model. The new select- $\mu$ op instruction was conceptually based on the  $\phi$ -function used by static-single-assignment (SSA). It allows the resolution of multiple definitions to be postponed to later stages of the pipeline, providing greater chance that a stall would not have to occur. To form the select- $\mu$ op instructions, the possible definitions of a use are needed. To this end they presented an augmented *Register Alias Table* (RAT), and use this to create the select- $\mu$ op instructions. Each set in the RAT represents all of the current potential definitions for each logical register. Each entry contains the renamed definition and the guarding predicate under which it was defined. The most recent definition guarded by a true predicate is the correct definition. Once this definition is determined, any further dependencies left to be reconciled could be eliminated from consideration. We simulated the hardware select- $\mu$ op mechanism used by Wang et. al. [24], and compare its results to our pure software Phi-predication.

## 3 Phi Predication Definition

In this section we present Phi-Predication. The Phi-Predication ISA has four important hardware constraints. First, all instruction register writes are unconditional to avoid the multiple-definition problem. This is the key property that distinguishes select predication from conditional-writer predication. Second, speculative memory operations that can cause exceptions or side-effects are predicated. Third, selection is based upon a predicate register instead of a general purpose register. A predicate register differs from a regular register in that it's only a single bit versus full 64 bits. Thus implementing a read from a predicate register file is simply a wide mux (multiplexor), while a 64 bit predicate register requires a register port. The following subsections describe

the instructions needed to support Phi predication and how to use the instructions for compiler transformations. Fourth, to keep exception handling synchronous and reported only on the predicated true path, we need to carefully qualify exception reporting. This is handled through special memory and floating-point operations.

### 3.1 Phi-Predication Instructions

Based on our hardware constraints, our Phi-predication implementation requires four different predicated classes of instructions to efficiently form predicated regions. Table 2 shows all of our predicated instructions, all other instructions (e.g., general purpose instructions) are not predicated.

PHI
phi r32=(p1),r33,r34
phi r32=(p1),100,r34
fphi f32=(p1),f33,f34
MEMORY
ld r32=[r33],(p1)
st [r32]=r33,(p1)
ORP
orp p1=p2,p3,p4
UNCONDITIONAL COMPARE
cmp.eq.unc p1,p2=r3,r4,(p3)
cmp.gt.unc p1,p2=r3,r4,(p3)
cmp.ge.unc p1,p2=r3,r4,(p3)

Table 2: Assembly Opcodes

The first class of instruction is our PHI operations to guard register data-flow. The phi instruction behaves like the select instruction proposed by Lowney et. al. [14] except that it uses a predicate register to do the selection. For the integer form PHI, we provide two types: one has two input register operands, and the second has an input operand plus immediate to optimize away small constants. For floating point FPHI, we only use a two input register operand instruction.

The second class of instructions are MEMORY operations, whose exceptions and memory updates are conditioned on a predicate register. Loads and stores fall under this class, but data-prefetch instructions do not. If a load instruction’s guarding predicate is defined as false, it will have a value of zero written to its destination register. The only conditional-writing instruction in our ISA is the predicate store, which conditionally updates memory based upon the outcome of the register. Therefore, out-of-order load-store forwarding has the potential to suffer from the multiple-definition problem, but this should not be a performance issue for memory disambiguation architectures like Store Sets [4].

The third type of instruction is a Boolean OR of predicates called the ORP. It applies Boolean OR operations to three input predicate registers, and writes an output. As seen in the next section, this ORP is used to combine predicates representing the control flow to a join block. If all predicate inputs are false then the predicate definition is set to false, otherwise it is set to true.

The fourth class of instructions are predicate definition compares (UNCONDITIONAL COMPARES) whose predicate values are always updated. If the guarding predicate is false, then the predicate definitions are set to false.

### 3.2 Transformation

We now describe if-conversion compiler transformations supported by our predicated instruction set, as shown in Figure 2. The first two examples (class 1 and 2) show the transformation of predicated if-then-else (hammock) regions of code. The phi instructions are used to select the result from the two paths of execution. Our compiler uses SSA [6] to optimally find the point to insert the phi instruction. In our simple “if-then-else” (hammock) control flow, it is the join block. Note the phi instruction in the first example uses the constant from the assignment on one path, thereby eliminating one of the move instructions. We found this to be an efficient *move combining optimization*. This optimization can sometimes reduce the instruction count for a region to equal or below that of conditional-writer predication.

The third transformation shows the predication of two nested if-then statements. When program execution goes through the first compare, but not the second, the second unconditional compare predicate assignment is forced to false.

The fourth example transformation shows where there is a join of two predicate definitions. This shows the use of the ORP instruction, which computes the union of the entering edge predicates. Only if there is a use of the join block predicate do we materialize the ORP.

### 3.3 Phi-Chaining Issues

Phi-predication has the potential to lengthen the dependency path through a region of code and increase the instruction count due to inserted Phi-operations. With a single simple PHI-operation found commonly in an if-then-else, the static schedule of Phi-predication is equivalent to conditional-writer predication or even perfectly branch predicted control flow. However when there are more than two edges reaching a join point (and the edges contain different register updates), a sequence of Phi-operations is needed to evaluate the different updates. This sequence is a Phi-chain. To avoid this serial evaluation penalty, the compiler carefully forms predication regions that have minimal length Phi-chains.

### 3.4 Handling Exceptions

Since normal instructions are not guarded on predicates, when a normal instruction executes it is not clear if it is on the correct path of execution until its dependencies (direct or indirect) are consumed by a phi instruction. Therefore, to deal with speculative floating point exceptions, an exception will write a NaN (Not-A-Number) as is done in [10]. This would silently propagate through speculative FP-ops until after a specifically marked phi instruction (directly or indirectly) consumes the register. If the register is still poisoned by a NaN and it is chosen then the exception will be pro-

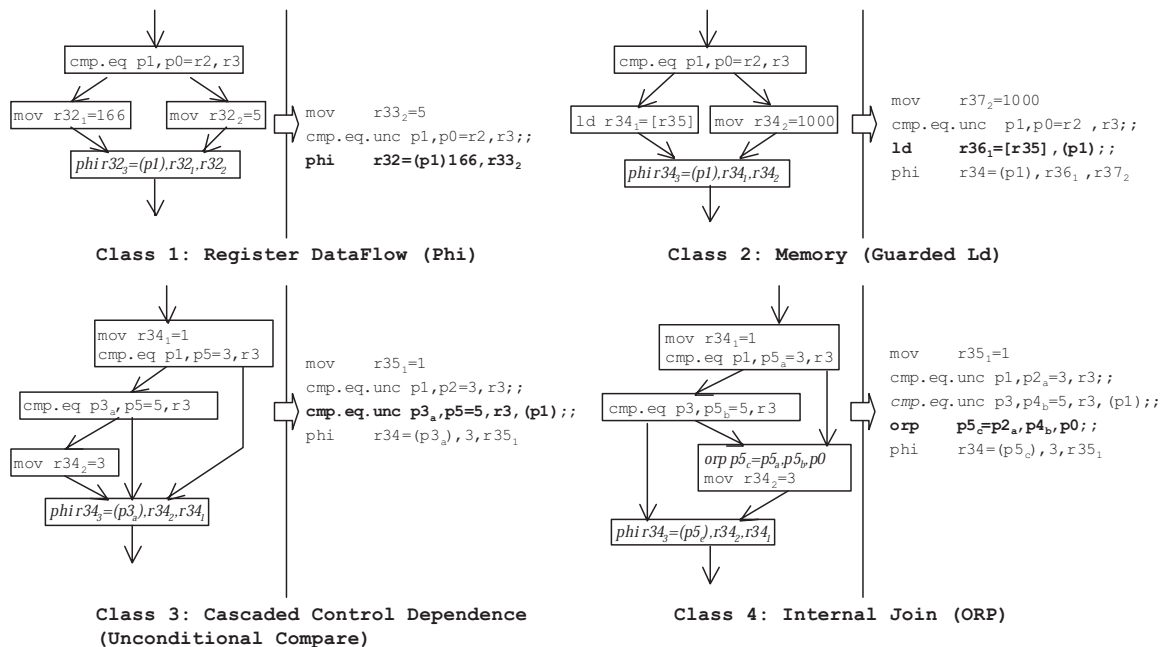


Figure 2: Four Classes of Select Predication Compiler Transformations. The control flow on the left is transformed to the if-converted code on the right. Repredicative instructions of the class are in bold font. Italicized instructions are introduced by if-conversion.

cessed, otherwise the propagation will stop and the exception will be ignored.

## 4 Phi Prediction Compilation

This section provides the compiler details and algorithms we used to form predicated regions, where to insert predicate definitions, phi operations, and predicate or-predicate operations.

### 4.1 Phi-Predication Strategy

We implement Phi-predication by modifying the pre-existing Intel IA64 compiler “predicator” pass. A general description of the IA64 compiler code generator is found in [1].

The following section describes the strategies the Phi-predication phase takes to generate useful if-converted code. For region formation, we use a basic block region picker to find single entry and single exit regions, without any returns, calls, or any system instructions with non-register or memory side-effects. Next we filter the region using heuristics or profiling information to increase the likelihood of profitability. If-conversion will gain performance when a mispredicting branch is eliminated. It loses performance when the combined path exceeds the original path.

When if-converting a region, the predicator starts by initializing each block with a unique predicate. To achieve this we use a modified version of the RK algorithm described in [16]. The RK algorithm determines the guarding condition assigned to each basic block along with the boolean tests required to calculate the guard predicates. We use a modified Single-State-Assignment based on [6] to make efficient decisions for PHI-op insertion along the dominance frontier of a block.

In the rest of this section, we describe more precisely how we compute the predicates and place the Phi-predication instructions. The following is the summary of steps our algorithm takes:

1. Scan the control flow graph for candidate acyclic regions and use heuristics to determine region formation suitability (Section 4.2).
2. Determine predicate defining instruction location in control flow, and determine ORP instruction join block locations (Section 4.3).
3. Phi-op generation using modified SSA, and perform combining and redundancy phi optimizations (Section 4.4).
4. Convert control dependent load and stores into phi form.
5. Collapse acyclic control flow region into single block.

### 4.2 Region Formation

Our predication pass constrains the selected region for if-conversion to simplify our analysis of Phi-predication. Our algorithm operates on a flow graph with a single entry and a single exit edge. All blocks end with conditional branches or unconditional branches. This allows for at most two successor edges for any block, and prohibits calls, returns and indirect branches inside predicated regions. We also prohibit system instructions or other instructions with system side-effects inside of predicated regions. In addition, speculative or post-increment memory operations are also prohibited. For this paper, these constraints allowed us to create an efficient algorithm for Phi-predicated if-conversion. Interesting potential future work deals with relaxing these constraints (e.g., considering multiple exit regions).

Our goal during region selection is to avoid the branch mispredict penalty while minimizing any other penalty. If-

conversion can hurt performance in several ways if we are not careful. First, the schedule of the shorter paths can be penalized by the longest path. Second, with Phi-predication there are PHI-op(s) inserted on the critical path for the schedule. Third, the combined paths might be resource constrained, lengthening the schedule particularly for long paths. To guide the formation of regions we apply profitability or profiling heuristics as described next.

#### 4.2.1 Short Static Region Heuristic

Our strategy for static region formation is to focus on if-converting short regions to reduce the length of phi-chains, and minimize resource contention. The overriding heuristic to guide static formation is that we do not form regions that take more than four cycles to schedule. This limits the potential penalty associated with executing the short path in the region. In this case, the schedule difference between the shortest and longest path cannot be longer than three cycles.

#### 4.2.2 Profile-Based Heuristic

We also examine using branch profiling information to guide region formation. Using profiling information from the training input set, we are able to obtain a more precise estimate of the branch misprediction penalty and use this to guide the profitability of performing if-conversion.

We first compute the schedule of all paths from entry to the exit join, and the combined resource path length, picking the longest schedule from these estimates. We then gather profiling information to compute the probability of the branch being mispredicted and the probability of traversing each path in the region. Our branch misprediction probability is derived from branch profiling that simulates the branch predictor used in the underlying simulated architecture.

The equation below compares the benefit from eliminating the mispredicting branch against the schedule penalty for collapsing the regions. The variables on the left side are:  $p_b r_i$  the branch execution probability for branch  $i$ ,  $M$  the mispredict penalty, and  $p_{mi}$  the mispredict probability for branch  $i$ . On the right side,  $p_j$  is probability of traversing that path,  $H_j$  the schedule height of path  $j$ , and  $H_l$  the longest schedule in the region.  $RgnBr$  is the set of branches, and  $RgnPath$  is the set of paths in the region. The intuition behind this equation is that the number of cycles saved by removing the branch misprediction penalty needs to outweigh the number of additional cycles it will take to fetch/execute the shorter paths in the resulting predicated region. We perform the test at the exit block of the region because it is the confluence of all the paths through the region from the entry block.

$$\sum_{i \in RgnBr} p_b r_i p_{mi} M > \sum_{j \in RgnPath} p_j (H_l - H_j)$$

Intel's IA64 compiler also uses path profiling information as a carefully tuned heuristic for selecting regions. We do not make use of this information in our region selection heuristics.

### 4.3 Inserting Predicate Definitions in a Region

Now that we have selected our if-conversion regions, conditional branches need to be removed from the instruction stream and Phi-predicated instructions inserted. We build on the RK algorithm [16] to find the predicate definition and use points. We now describe how we determine where to insert the predicate definitions and what predicates should guard each original basic block in the region. We first define the control dependence relationship [6] between edges and blocks in the regions to determine where to insert the predicate definitions.

**Definition 1** *Block Y is control dependent on directed edge  $E=(X,Z)$ , if there exists a path from X to Y traversing Z, such that all blocks on any path from X to Y are post-dominated by Y, and X is not post-dominated by Y.*

If blocks share the exact same control dependence relationships, they will be guarded by the same predicate. To facilitate this, we set up an equivalence class of blocks based on control equivalence as defined in [16]. Block X is control equivalent with block Y iff they have the same set of immediate control dependence edges. We write the equivalence class of block X as  $X_{ce}$ . Another perspective on these relationships are in [7], which also provides efficient algorithms.

To allow the result of an operation to be accessible to all blocks in a control equivalence class, we need to find the first block that is executed, and insert that operation at the entry to the block. We find this *cap* block by finding the block in the control equivalence set that dominates all other blocks in the set.

To determine where to insert predicate definitions, we use the RK algorithm to find all control dependence edges into a given set of control equivalent blocks. The RK algorithm [16] states that each control edge in C is either the taken edge (called +y in RK) or the not-taken edge (called -y in RK) from a conditional branch, where C is the general set of all possible original control edges in the region. Therefore, for a set of equivalent blocks:

$$CD(X_{ce}) = \{ \pm y \in C : \text{Blocks in } X_{ce} \text{ are control dependent on } +y \text{ or } -y \}$$

For each edge found in  $CD(X_{ce})$  our algorithm inserts or confirms a predicate definition representing that edge's path. For  $CD(X_{ce})$  with no control edges, those blocks will have no guarding predicate, since they will always be executed. To illustrate CD, consider two blocks A and B, where B is a successor of A through the taken edge. If  $CD(B) = +a$ , this states that the branch in A determines whether B will execute.

#### 4.3.1 Finding Guarding Predicate for Control Equivalence Blocks

Now we need to figure out the predicate defining statements needed. We can think of predicates as having two different

uses. One purpose is to guard the execution of a basic block. A second purpose is to represent traversal through a control-dependence control-flow edge. Often these predicates map to the same value, but not always. The original RK algorithm [16] does not make this distinction because that work uses conditional-writer predicate definitions to implement a predicate join. Since we have always-write predicate definitions, we must instead use the ORP instruction. This facilitates determination of control edge predicates and block guarding predicates (for potential joins). We now describe how we determine this mapping for Phi-predication.

Our algorithm initially assigns to each equivalence class block  $X_{ce}$  a guarding predicate  $p_g$ . We also define a function  $r$  for an equivalence class to represent its final guarding predicate. Initially  $r(X_{ce}) = p_g$ . Other potential predicates will come from control edges. When the conditional branch is removed, it is replaced with two predicate definitions  $p_c$  and  $p'_c$ . If the original branch condition test is  $t_y$ , then  $p_c = t_y$  and  $p'_c = \neg t_y$ . There is a mapping function  $k$  from the control edge predicate  $p_c$  to its control edge  $c$ , is  $c = k(p_c)$ .

The original RK algorithm, computes a direct mapping between the guard predicate and the control predicate. In our algorithm, we need to do this mapping differently, since we have no conditional-writer predicate definitions. The following function summarizes how to set the final guarding predicate:

$$r(X_{ce}) = \begin{cases} true & : \text{if } |CD(X_{ce})| = 0 \\ p_g = p_c \text{ where } c = k(p_c) & : \text{if } |CD(X_{ce})| = 1 \\ \text{and } c \in CD(X_{ce}) \\ p_g \text{ where } p_g = & : \text{if } |CD(X_{ce})| > 1 \\ \bigvee \{p_c : c = k(p_c) \text{ and } c & \\ \in CD(X_{ce})\} & \end{cases}$$

If the control equivalent blocks in  $X_{ce}$  have no control edge, then the guarding predicate is the true predicate, since the blocks will always be executed once the region is entered. If there is only one control edge, we can directly set the guarding predicate equal to the predicate definition for the controlling edge ( $p_g = p_c$ ). If there are multiple controlling edges, then there can be multiple predicate definitions. To address this we insert an ORP predicate operation on the cap block for the control equivalence class. We then set the block guarding predicate to the result of the ORP.

In addition, any conditional memory operations with architectural side effects like loads and stores needs to use the phi-speculative form. Other memory operations like prefetch instructions need not be converted. We determine if a memory operation is conditional by checking the instruction's CD set. Given the block X, if  $CD(X)$  is non-empty we need to use the phi version of the memory operation, otherwise leave it alone. We also convert all compares to unconditional form in the region.

#### 4.4 PHI-Op Generation

Next our algorithm finds where to insert each phi operation, what its operands are, and what its PHI-op selector predi-

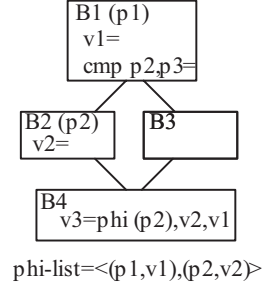


Figure 3: When a PHI-op is inserted at B4, we would like its selector to come from B2 instead of B1.

cates are. Our algorithm builds upon SSA described in [6]. SSA creates a compiler internal representation where every variable use has a unique definition. Confluence of the variable's different definitions at control flow joins are computed through  $\phi$ -functions. These functions then return a new instance of the variable, conforming to the single assignment rule. Our Phi-predication uses a variation of the  $\phi$ -function representation called phi-lists, that are later converted into our PHI instructions.

We start by finding the potential candidate phi-lists. For any live variable at the entry to a block, there might be a phi-list associated with it. Each phi-list is a sequence of instances of some variable. The instance has associated: its defining block, that block's guard predicate and topological number. There are two important differences between the phi-lists we use and those defined in the original SSA. First, each instance in the phi-list has additional information, especially the guard predicate of the block where the instance is defined. The second difference is we topologically order these definitions in a phi-list to determine the correct order in which to insert phi instructions. Topological ordering describes the degree of "closeness" meaning how close the definitions reaching the phi operation is. When we evaluate a sequence of phi operations, we want to start from the farthest in terms of topological order (that becomes the default value) to the closest in terms of topological order.

The need for the topological ordering constraint is best seen through an example. Consider Figure 3 where there is an if-then-else with an assignment to variable instance  $v1$  (variable  $v$ ) in block B2 on one path, and an assignment  $v2$  at the entry block B1 to the region. A PHI-op will be generated at the exit block B4. We denote the block guard predicate for B1 as  $p1$ , and B2 as  $p2$ . Clearly if we set the predicate  $p1$  to be the PHI selector predicate, it wont be able to choose between  $v1$  and  $v2$  correctly when execution goes through B3 in one case, and B2 in another. By using the topological ordering constraint, we would correctly select  $p2$  as the PHI-op selector predicate.

One complication for phi-list generation, is that the phi-list defines a new variable instance. Because we want to capture all possible variable definitions at the phi-list inputs, we need to carefully generate phi-list such that all predecessor

blocks phi-lists have already been evaluated. If we process blocks in topological order in our acyclic region, we can satisfy this second constraint.

Our algorithm walks over the control flow region in topological order looking for variable assignments. When an assignment is found, all blocks that are in the dominance frontier [6] of the block have their phi-lists updated to account for the assignment.

Next we convert each phi-list into Phi instructions, ordered in topological order. During PHI-op insertion, we take two adjacent instance pairs off the phi-list, and use the closest (in topological order) predicate as the selector operand of the PHI. If there are more than one PHI-op to be inserted, we use a temporary result register.

We can eliminate extra move operations by using instruction combining optimizations. One optimization is to subsume a MOV source register into the PHI source register. Another optimization is to take a MOV source constant value, convert the PHI to the immediate form, and subsume the constant into the immediate. As MOV instructions are common, this optimization is fairly prevalent.

## 5 Methodology

We give our results from Spec2000 Int and FP benchmarks, using a derivative of the Electron Intel IA64 C-language compiler to implement our algorithms. Our results are for the following SPEC 2000 programs: 164.gzip, 175.vpr, 177.mesa, 179.art, 183.equake, 186.crafty, 197.parser, 256.bzip2, and 300.twolf. The benchmarks were compiled with “-O3” compile flags without interprocedural optimizations. We present all SPEC 2000 programs that would compile and trace properly with our modified tools to provide a fair performance measurement.

Results are provided for forming predicated regions using static heuristics, or branch misprediction profiling. Results with *Br* in their name use branch profiling, all other results use static heuristics. In addition, results denoted with *Path* in their name use path profiling to apply wavefront scheduling after regions are formed. For the speedup results presented we used the “reference” inputs. We used the “train” inputs to generate path and branch profiling information. Fastforward information was derived from SimPoint [20].

The compiler and the rest of the “compilation” tool-chain was modified to create Phi-predication binaries. We developed a trace based simulation methodology where we separate functionality from performance simulation.

Given a binary, our tracing tool functionally emulates the non-native phi instructions, generating a trace to be consumed by our simulator. We target generating a trace of 300 million instructions, with some variation due to scheduling differences between the different executables. Since the different binaries (Phi-predicated, IA64 predication (Cond-

Writer Pred), IA64 non-predicated, select-op only, branch profiled, and branch+path profiling) have different instruction mixes, we must be careful to make sure we simulate the exact same segment of execution. Start points were approximated from SimPoint data on the representative regions. We use the function entry that contains the representative region as our start point. To precisely find equivalent end points between these binaries, we use a trace comparison tool that finds the last equivalent function call or return site on all of the traces for a given program and input. *Start* and *end* points are given in Table 3. The traces provide a representation of the functional dynamic execution behavior of a program, which we then input to our performance simulator to obtain cycle time results. Variation in trace length can be seen in Figure 4 as compared to compilation with no predication.

We modified SimpleScalar-3.0 [2] to consume IA64, and Phi-predication ISA instructions. The architecture model we used is summarized in Table 4. We model in detail Itanium features such as rotating registers, predication, and register stack engine behavior. In particular, we carefully model Itanium resources and instruction latencies based on [9] to provide a meaningful target for the compiler, as the compiler assumes this, and schedules for this. Our out-of-order model includes an optimization called early predicate evaluation, that squashes falsely predicated definitions early in the pipeline if the predicate value is known in the rename stage of the pipeline. This optimization is applicable to almost all instructions for IA64 predication, and only to load and store instructions for Phi-predication. Table 4 shows the number of pipeline stages we simulated between rename and execute.

To make fair comparisons between different compilations, we compile Phi-predication and IA64 conditional writer predication with the same if-converter region picker. Thus the same predicated regions are used between the two ISAs. Similarly the same traces are used between the in-order, and the out-of-order simulations.

Benchmark	Start Point	End Point
164.gzip (random)	longest_match	ct_tally
175.vpr (place)	get_non_updateable_bb	get_bb_from_scratch
177.mesa	general_textured_triangle	read_min
179.art	match	reset_nodes2
183.equake	phi0	smvp
186.crafty	SwapXray	GenerateCheckEvasions
197.parser	region_valid	magic
256.bzip2 (graphic)	spec_putc	fullGtU
300.twolf	ucxx1	new_dbox_a

Table 3: Trace Characteristics: Because predication may introduce additional instructions in the trace, we synchronized execution between known start and end points. () further specifies input if multiple.

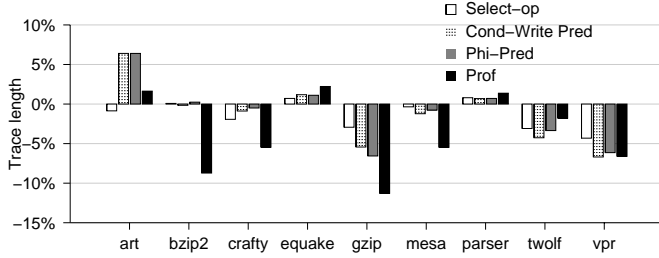


Figure 4: Benchmark tracelength difference

General Machine Model	
Fetch Width	Up to 2 Bundles
Issue Width	6 instructions
Function Units	Itanium latencies, resource units
Reorder buffer	RUU: 256, LSQ: 128
L1D	64KB, 4 way, 32B Block, 2 cycle latency
L1I	64KB, 4 way, 32B Block, 1 cycle latency
L2 Unified	4MB, 4 way, 64B Block, 12 cycle latency
DTLB	128 entry, 30 cycle miss penalty
ITLB	64, 30 cycle miss penalty
Memory	300 cycle latency
Branch Prediction	8K meta chooser between gshare (8K entry) and bimodal table (8k entry); 32 Return Address Stack; 512 BTB; 12 cycle miss penalty
RSE	Lazy spill and reloads, model register stack
Software Pipelining	Model rotating registers
PHI-op & select- $\mu$ op	1 cycle latency
Model Specific	
In-order	Br mispred delay: 9 cycle
Out-order (12)	Br mispred delay: 12 cycle Pipeline stages between rename and exe: 5
Out-order (30)	Br mispred delay: 30 cycle Pipeline stages between rename and exe: 10

Table 4: IA64 SimpleScalar: This table describes the general characteristics of our trace-driven simulator.

## 6 Results

In this section we provide results for the code generated by the compiler, and provide a comparison of Phi-predication to IA64 predication for an in-order and out-of-order processor.

### 6.1 Compiler Statistics

In analyzing the compiler’s Phi-predication effectiveness, we wanted to observe what kind of regions our region-picker finds, how effective the instruction set is at accepting regions, and the effectiveness of our phi optimizations. One measurement of effectiveness is the distribution of the size of the selected regions by the if-converter region picker, as seen in Figure 5. The Y-axis represents the total percentage of if-converted regions summed over all programs. The total number of if-converted regions was 2720. The X-axis represents the number of basic blocks in the given region. That results show that small “if-then-else” regions of 3 to 4 basic blocks comprise 72% of the regions. Small region bias is due in part to our static region heuristic. The graph also shows that the largest region predicated consists of 19 basic blocks, which was in `twolf`. This region represents an example of complex control flow.

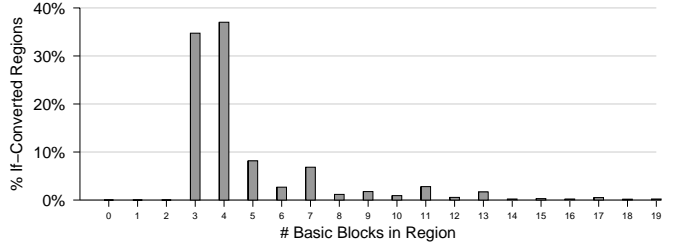


Figure 5: Phi-Predication If-Convert Region Size Distribution: X-axis measures number of basic-blocks in region. Y-axis represents the percentage of all if-converted regions.

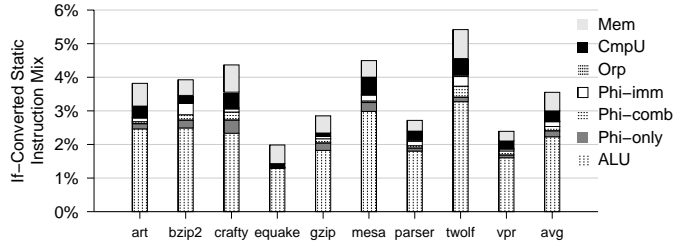


Figure 6: Percent of static instructions in if-converted regions. Instructions broken down into the different instruction types.

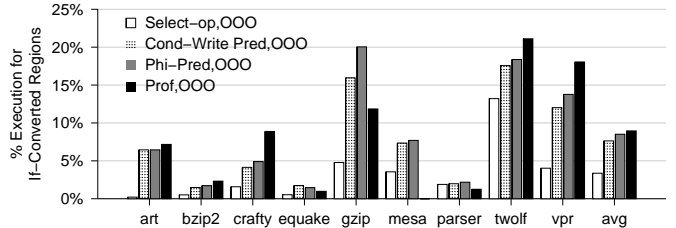


Figure 7: Percent of execution accounted for by if-conversion regions.

Figure 6 shows the percent of static instructions in if-converted regions for each program, and the breakdown as to the type of instruction’s used in that region. The instruction types are those described in Section 3.1. The `comb` bar segment represents the target of a MOV register combining optimization, and `imm` represents the target of MOV constant combining. Though not apparent from the graph, 60% of PHI-ops take part in the combining optimization, as described in Section 4.4. We see on average that only 3.6% of the static instructions are in if-converted regions after Phi-predication. Figure 7 shows the percent of dynamic execution accounted for in these regions. This shows that on average 8.5% of execution is in if-converted regions, but this varies greatly from 1.7% for `equake` to 20.0% for `twolf`.

### 6.2 In-order Results

Our first set of results involves simulating an in-order, Itanium-like pipeline. The machine model has a minimum branch mispredict penalty of 9 cycles, and simulates stop bits and IA64 bundling behavior. We compare Select-op, Phi-

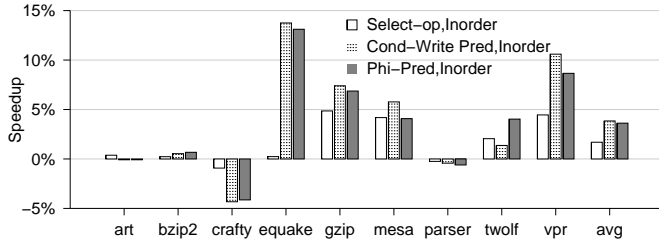


Figure 8: In-order Speedup normalized against no-predication with a minimum branch mispredict penalty of 9 cycles. Phi-Predication and IA64 Conditional-Writer predication have the same if-conversion regions. The arithmetic average is 3.6% and 3.8% respectively.

predication, and Conditional-Writer predication in Figure 8. Select-op provides a comparison to the minimal amount of prediction support, which uses only unconditional compares to generate predicates and select operations to choose between two definitions (see Section 2.1). These results show three important points. First, we see that conditional writer predication has a slightly greater average speedup than Phi-predication for in-order: 3.6% for Phi-predication and 3.8% for conditional-writer predication. We expect some performance imbalance due to the penalty associated with long phi-chains, but the observed imbalance is quite small. Second, these speedups could be viewed as being in the same ball park with the results found in [3]. They found a 2% speedup on real hardware. Third conditional-move predication using Select-op predication gets only a smaller 1.7% speedup.

### 6.3 Out-of-order Results

For the in-order results, much of the benefit of eliminating the branch mispredictions via predication is hidden by the in-order execution constraints. When considering out-of-order execution, where instructions are allowed to execute as soon as their data-dependencies are fulfilled, there is more potential gain from Phi-predication. To account for the additional complexity of out-of-order, we increase the minimum branch mispredict penalty from 9 cycles to 12 cycles. As noted earlier, to get decent performance from IA64’s conditional-writer predication requires using hardware inserted select- $\mu$ op instructions. This provides hardware support for the multiple-definition problem as described by Wang et al. [24], which we label as Wang in the results.

We first examine the relative performance of the in-order vs out-of-order execution models. Figure 9 shows the normalized execution time between the in-order and out-of-order models both without predication, and with existing conditional-writer IA64 prediction.

Figure 10 shows the speedup of using Phi-predication and conditional-writer IA64 predication for the same regions on the out-of-order processor vs no-predication. The IA64 predication results uses Wang et. al’s [24] hardware select- $\mu$ ops optimization. We see an average increase in performance over no-predication of 5.8% for Phi-predication,

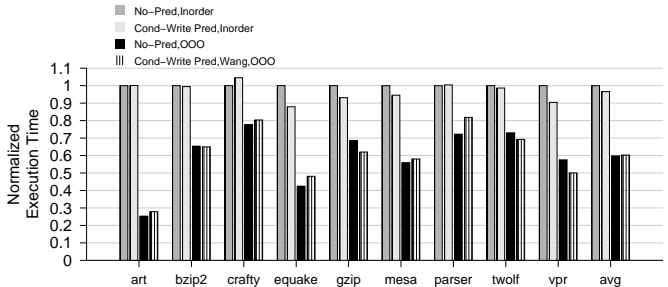


Figure 9: Execution cycle time normalized against no-predication in-order.

but only -0.9% for conditional-writer IA64 predication, even with the select- $\mu$ ops. Much of the performance divergence from the dynamically generated select- $\mu$ ops in the Wang model comes from insertion of the select- $\mu$ ops, which are not as precisely inserted as the PHI-ops using our compiler approach. The Select-op only compiler optimization results in an average 2.0% speedup.

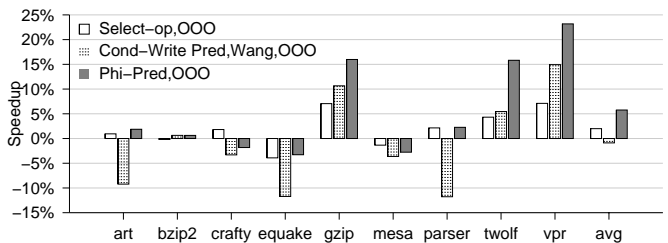


Figure 10: Percent speedup over an out-of-order processor using no-predication. These results use a 12 cycle minimum branch mispredict penalty.

To further examine the results, Figure 7 showed the percent of executed code in predicated regions. If we start from the premise that lots of if-conversion gains performance, then some benchmarks act somewhat as predicted, like gzip or twolf. Similarly benchmarks with very little if-conversion have very little performance change like bzip2. Unfortunately there are programs like mesa and equake that have negative performance with only modest amount of execution in if-converted regions. Some behavior can be explained by looking at branch mispredicts in Figure 11. This figure shows the absolute number of mispredicted branches using no-predication, using conditional-writer IA64 predication, and using Phi-predication. The results show that there is little reduction in the number of mispredicted branches for bzip2, equake or mesa, hence little benefit. For equake and mesa the scheduling penalty due to if-conversion must have negatively overwhelmed any small benefit. In comparison, the programs with the largest speedup see a significant reduction in the mispredictions like gzip and twolf. Note that conditional-writer IA64 and Phi-predication had essentially the same branch mispredict reduction.

In looking at future trends in microprocessor design,

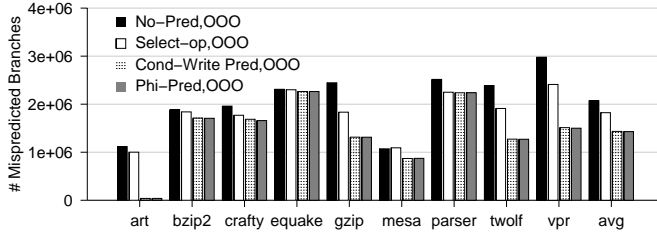


Figure 11: Total number of mispredicted branches for the out-of-order (12 cycle minimum mispredict) execution model.

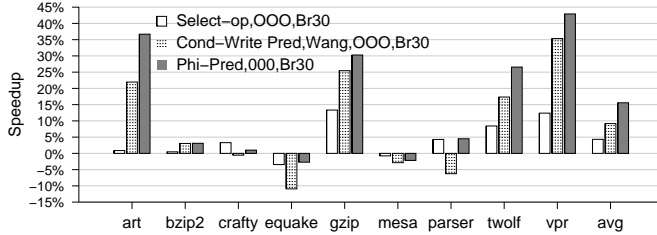


Figure 12: Out-of-order Speedup with a minimum branch penalty of 30 cycles as normalized against no-predication. The arithmetic average is 15.6%, 9.2%, and 4.3% for Phi-predication, conditional-writer, and select-op predication.

some designers believe that very deep pipelining is beneficial. For example Intel’s current Pentium 4 has a minimum 20-cycle branch mispredict penalty, and research supports building pipelines in excess of 40-stages [23]. Therefore, we also examined out-of-order results using a 30-cycle minimum branch misprediction penalty, shown in Figure 12. The results show an average speedup increase from 5.8% to 15.6% for Phi-predication. Writer-predication and Select-op predication also sees an improvement from -0.9% to 9.2%, and 2.0% to 4.3%.

## 6.4 Profiling Results

All of the results up to this point used static information to guide region formation. Figure 13 shows using branch profiling information to guide region formation. All results are

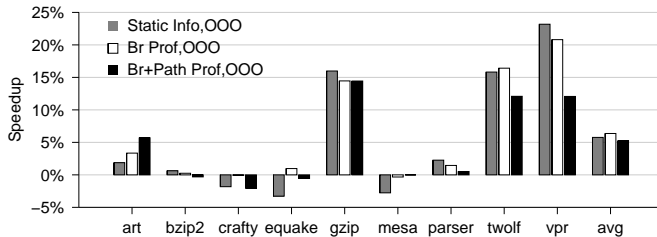


Figure 13: Profiling Information with Phi-predication. All results are out-of-order. The first two bars are normalized to baseline with no predication and without path profiling. The last bar is normalized to no predication with path profiling. These results are shown for a 12 cycle minimum branch misprediction penalty. The second bar shows results for branch profiling, and the last bar uses path and branch profiling.

normalized against no predication, with the first two bars without path profiling and the last bar with path profiling. First we consider branch mispredict profiling in selecting if-conversion regions. Branch profiling only has an average speedup of 6.4% versus short-static heuristic 5.8%. The results show that our branch profiling resulted in more conservative region formation, which prevented slow-downs for `bzip2`, `crafty`, and `equake`, but at the same time did not achieve as high of speedups for `gzip` and `vpr`. Tuning of our branch profiling heuristic is needed to stay conservative where predication will hurt, and to be more aggressive when predication can be beneficial.

We next consider path profiling [14]. The Electron compiler takes into account edge and basic block frequencies in multiple phases, but has the most impact in code scheduling. When using path profiling, the predicated regions are formed first using the branch heuristics, and then the path profiling information is used to guide wavefront scheduling after the regions are formed. Using both branch and path profiling achieves an average 5.4% speedup for 12-cycles branch misprediction. This is in comparison to the baseline architecture with no predication, and using path profiling for wavefront scheduling. These results show that even with aggressive profile guided scheduling Phi-predication can be beneficial.

## 7 Concluding Remarks

In this paper we describe the Phi-predication ISA and the compiler algorithms used to generate it, and examine its performance on a number of benchmarks for an in-order and out-of-order processor based on IA64. Phi-predication eliminates the register multiple-definition problem, eliminates predicated bypasses, and reduced opcode pressure over heavy-weight writer predication. Our Phi-predication ISA and compiler algorithm are able to handle large single entry and exit predicated regions for various sizes and complexity.

We found that Phi-Predication provided an average 3.6% performance improvement over no-predication for in-order processor. For an out-of-order processor with a minimum branch misprediction penalty of 12 cycles, the performance improvement over no predication was 5.8% on average. This contrasts with an average -0.9% slowdown with conditional-writer predication using hardware select- $\mu$ ops as described in [24], or only a 2.0% speedup with compiler-based select-op predication. When the misprediction depth is increased to 30 cycles the improvement increases to 15.6% for Phi-Predication. Overall, Phi-Predication provides a light-weight solution to achieve important improvements by eliminating hard to predict branches for an out-of-order processor.

## 8 Acknowledgment

Our work builds on Intel Electron compiler predictor that was written by William Chen. We thank the anonymous reviewers for their thorough and insightful comments. This

work was funded by National Science Foundation grant No. CCR-0073551 and a grant from Intel Corporation. We especially would like to thank Intel for providing the Electron compiler sources, and their assistance in using it.

## References

- [1] J. Bharadwaj, W. Chen, W. Chuang, G. Hoflehner, K. Menezes, K. Muthukumar, and J. Pierce. The Intel IA-64 Compiler Code Generator. *IEEE Micro*, 20(5):44–52, Sept 2000.
- [2] D.C. Burger and T.M. Austin. The SimpleScalar Tool Set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, Jun 1997.
- [3] Y. Choi, A. Knies, L. Gerke, and T.F. Ngai. The impact of if-conversion on branch prediction and program execution on the intel itanium processor. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 182–191, Dec 2001.
- [4] G. Chrysos and J. Emer. Memory dependence prediction using store sets. In *Proceedings of the 25th International Symposium on Computer Architecture*, June 1998.
- [5] R.P. Colwell, W.E. Hall, C.S. Joshi, D.B. Papworth, R.K. Rodman, and J.E. Tornes. Architecture and implementation of a vliw supercomputer. In *Supercomputer '90*, pages 910–919, Nov 1990.
- [6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
- [7] R. Cytron, J. Ferrante, and V. Sarkar. Compact representations for control dependence. In *ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 337–351, June 1990.
- [8] J. C. Dehnert, P. Y. Hsu, and J. P. Bratt. Overlapped loop support in the cydra 5. In *Architectural Support for Programming Languages and Operating Systems*, pages 26–38, April 1989.
- [9] Intel Itanium Processor Reference Manual for Software Optimization, November 2001. <http://developer.intel.com/design/itanium/downloads/245474.htm>.
- [10] IA-64 Application Instruction Set Architecture Guide, Revision 1.0, 1999.
- [11] M. F. Jacome, G. de Veciana, and S. Pillai. Clustered VLIW architectures with predicated switching. In *Design Automation Conference*, pages 696–701, 2001.
- [12] V. Kathail, M. S. Schlansker, and B. R. Rau. HPL PlayDoh architecture specification: Version 1.0. Technical Report HPL-93-80, HP Labs, Feb 1994.
- [13] R. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, Mar–Apr 1991.
- [14] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg. The Multiflow Trace Scheduling compiler. *The Journal of Supercomputing*, 7(1-2):51–142, May 1993.
- [15] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W. W. Hwu. A comparison of full and partial predicated execution support for ILP processors. In *ISCA*, pages 138–150, 1995.
- [16] J. C. H. Park and M. Schlansker. On Predicated Execution. Technical Report HPL-91-58, HP Labs, May 1991.
- [17] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Trowle. The cydra 5 departmental supercomputer: design philosophy, decisions and trade-offs. *Computer*, pages 12–35, January 1989.
- [18] M. Schlansker and B. R. Rau. EPIC: An Architecture for Instruction-Level Parallel Processors. Technical Report HPL-1999-111, HP Labs, 2000.
- [19] H. Sharangpani and K. Aurora. Itanium processor microarchitecture. *IEEE Micro*, 20(5):24–43, Sept–Oct 2000.
- [20] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Languages and Operating Systems*, October 2002. <http://www.cse.ucsd.edu/users/calder/simpoint/>.
- [21] J. Sias, H. Hunter, and W. Hwu. Enhancing loop buffering of media and telecommunication applications using low-overhead predication. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, December 2001.
- [22] R. L. Sites and R. T. Witek. *Alpha AXP Architecture Reference Manual: 2nd Ed.* Digital Press, Boston, MA, 1995.
- [23] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *29th Annual International Symposium on Computer Architecture*, pages 25–36, May 2002.
- [24] P. H. Wang, H. Wang, R. M. Kling, K. Ramakrishnan, and J. P. Shen. Register renaming for dynamic execution of predicated code. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, February 2001.