Using Predicate Path Information in Hardware to Determine True Dependences

Lori Carter Brad Calder

Department of Computer Science and Engineering
University of California, San Diego

{lcarter,calder}@cs.ucsd.edu

ABSTRACT

Predicated Execution has been put forth as a method for improving processor performance by removing hard-to-predict branches. As part of the process of turning a set of basic blocks into a predicated region, both paths of a branch are combined into a single path. There can be multiple definitions from disjoint paths that reach a use. Waiting to find out the correct definition that actually reaches the use can cause pipeline stalls.

In this paper we examine a hardware optimization that dynamically collects and analyzes path information to determine valid dependences for predicated regions of code. We then use this information for an in-order VLIW predicated processor, so that instructions can continue towards execution without having to wait on operands from false dependences. Our results show that using our Disjoint Path Analysis System provides speedups over 6% and elimination of false RAW dependences of up to 14% due to the detection of erroneous dependences in if-converted regions of code.

Categories and Subject Descriptors

C.1.3 [**Computer Systems Organization**]: Processor Architectures – Other Architecture Styles

General Terms

Performance

Keywords

Dependence Analysis, Path Analysis, Predicated Execution

1. INTRODUCTION

A feature of the Explicitly Parallel Instruction Computing (EPIC) architecture is its support for predicated execution. Predicated execution in the form of if-conversion [5, 15] removes hard-to-predict branches by combining both paths of a branch into a single path. In doing so, definitions of the same logical registers (originally from different paths) are intermingled. This makes data dependence analysis significantly harder. Without the appropriate predicate sensitive analysis, dependency assignments must be very conservative, ultimately including dependencies that are not required.

The EPIC philosophy is that the compiler should handle most of the dependence analysis and scheduling in order to simplify the processor, and at the same time the compiler has a broader view of the code [13]. In the case of the Intel Itanium (the first implementation of the IA64 ISA), a scoreboard is used by the hardware to make decisions on instruction dependencies. While some of the independence information can be encoded by the complier into the VLIW instruction grouping or *bundle* and passed on to the architecture, much of it will have to be re-calculated by the hardware without the benefit of predicate relationship information.

In this paper, we describe a Disjoint Path Analysis Architecture that allows us to re-create predicate relationship information in hardware. We show that this architecture can be used to decrease the number of data dependencies that are conservatively enforced for the current Itanium IA64 implementation. If the predicate relationship information indicates that a definition's path is disjoint from the use, the data dependence is not assigned. This means that the definition was on a disjoint path. Our results show that up to 14% of the dependencies in if-converted regions can be removed for the Itanium model, yielding an improvement in IPC of up to 6% for these regions.

2. MULTIPLE PATHS MEANS MULTIPLE DEFINITIONS THAT RESULT IN STALLS

Predicated execution is a feature designed to increase ILP and remove hard-to-predict branches. Machines such as the Intel Itanium with hardware to support predicated code include an additional set of registers called predicate registers. The process of predication replaces branches with compare operations that set predicate registers to either true or false based on the comparison in the original branch. Each operation is then associated with one of these predicate registers (the operations *guarding predicate*). In general, the operation will be committed only if its guarding predicate is true. This process of replacing branches with compare operations is called *if-conversion* [5, 15].

An example of if-converting a set of basic blocks into a predicated region can be seen in Figures 1 and 2. Figure 1

ICS'02, June 22-26, 2002, New York, New York, USA.

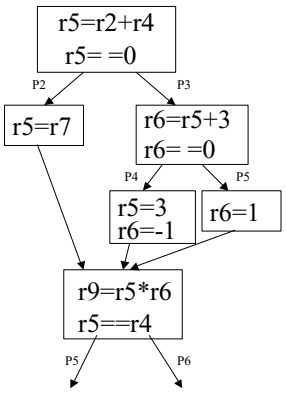


Figure 1: Original Control Flow Graph

```
      add r5=r2,r4
      cmp P2,P3 =r5,0
(P2) mov r5=r7
(P3) add r6=r5,3
(P3) cmp P4,P5 = r6,0
(P4) mov r5=3
(P4) mov r6=-1
(P5) mov r6=0
      mult r9=r5,r6
      cmp P5,P6=r4,r5
```

Figure 2: If Converted Version

```
[1]      add r5=r2,r4
[2]      cmp P2,P3 =r5,0
[3] (P2) mov r5=r7
[4] (P3) add r6=(r5[1]or r5[3]),3
[5] (P3) cmp P4,P5 = r6,0
[6] (P4) mov r5=3
[7] (P4) mov r6=-1
[8] (P5) mov r6=0
[9]      mult r9=(r5[1]or r5[3]or r5[6]),(r6[4] or r6[7] or r6[8])
[10]     cmp P5,P6 = r4, (r5[1] or r5[3] or r5[6])
```

Figure 3: If Converted showing multiple definitions of same register

shows the original control flow graph with 3 possible paths to the final block shown in the region. Figure 2 shows the if-converted code with the branches replaced and the 3 paths effectively combined into one.

As already mentioned, the benefits of if-conversion include the removal of hard to predict branches, and increased possibilities of finding *instruction level parallelism* (ILP). Normally, when a branch is encountered, the processor predicts the next address from which to fetch instructions to continue execution. There can be a large penalty if the prediction is incorrect. Predicated execution allows for a third possibility, executing both paths of the branch. Predication can also increase ILP because the predicated region can provide a larger pool from which to find independent instructions.

However, the process of combining multiple paths into one makes data dependence analysis significantly harder. Multiple paths containing definitions of the same architectural registers are intermingled. In the control flow graph in Figure 1, there are 3 definitions of `r5`, each occurring in a different basic block. However, in the if-converted code, the 3 definitions are in the same predicated scheduling region. When the compiler or hardware tries to set up dependencies, the reaching definition for a use is not necessarily the last definition.

It is clear from the control flow graph in Figure 1, that the definition of `r5` made by the `mov` instruction `r5=r7` could not be the definition used by the instruction `r6=r5+3`. The definition and use in this case are on completely separate (*disjoint*) paths. However, in the if-converted code (Figure 2), the `mov` provides the most immediate prior definition. If the hardware has a dependence detection method telling it that `P2` and `P3` are disjoint, a use of `r5` will only have to wait on the correct definition. Otherwise, both of the 2 previous definitions of `r5` in the if-converted region must be considered as possible definitions. It would not be until it was determined that the guarding predicate of one of the defining statements is false that the dependence could be broken.

The `mult` instruction has multiple possible reaching definitions of `r5` that are all valid as shown in Figure 3. It could not be determined statically which of the definitions reached. It would instead depend on the particular execution. If predicate registers `P3` and `P5` were defined as true, the very first definition of `r5` (`r5[1]`) would be the definition that reaches the use of `r5` in the `mult`. This is because it would be the only definition guarded by true, or the only definition on the taken path. However, if `P2` is true, there would be 2 definitions (from instructions r5[1] and r5[3]) along the way guarded by true. Both of these are valid definitions on the path. However, only the definition that is closest to the use and along the correct path provides the definition. The problem in data dependence analysis is determining which is the closest definition on the same path as the potential use.

2.1 Effect of Extraneous Definitions on the Hardware

In the baseline Itanium model, dependency relationships between a producer and consumer register, where at least one is predicated, cannot be handled by the hardware until the predicate value has been resolved. To accommodate this in Itanium, the producer of the predicate register and a potential consumer of a general purpose register guarded on that predicate must be scheduled 2 cycles apart from each other [4]. This holds true whether the predicate register has a value of true or false.

Consider the following code segment:

```
(1)       cmp P4,P5 = r8,r5    cycle 0
(2) (P4)  ld r7=[r5]           cycle 1
(3) (P5)  add r6=r7,3          cycle 2
```

Based on the current Itanium implementation, these statements must execute as shown above in their corresponding cycles. Statement 3 is a potential consumer of statement 2, so the architecture enforces a potential dependency between the two instructions. The dependency will be broken by the scoreboard when it is determined that either `P4` or `P5` has a value of false. However, as described in [4], the producer of the predicates and the potential consumer of the general register must be scheduled at least 2 cycles apart to allow time for this determination to be made. Under certain circumstances this latency is greater that 2 cycles.

In the above example, if the hardware can accurately determine that the predicates guarding instructions 2 and 3 are disjoint, then it can allow those two instructions to be scheduled and executed in the same cycle. This potential savings is the benefit exploited by the Disjoint Path Analysis architecture presented in this paper.

The Itanium is fully *score-boarded* to allow for real-time decisions to be made about instruction execution [13, 16]. The main function of the scoreboard is to determine when all dependencies are resolved and to enforce WAW hazards. As mentioned, the mechanism is capable of breaking dependencies when either the producer or consumer guarding predicate is evaluated to be false. For the scoreboard, many dependencies have to be recalculated by the hardware, and we examine incorporating our Disjoint Path Analysis architecture into the Itanium scoreboard to eliminate false dependencies (as described above) before the predicate definitions are resolved.

3. RELATED WORK

Both compiler and hardware approaches have been proposed for handling multiple definitions.

3.1 Predicated Multi-path Compiler Analysis

Gillies et.al. and Schlansker et. al. [12, 17] presented the use of the Predicate Query System(PQS). This system uses a predicate partition graph to statically describe disjointness. A definition and a use that originated from disjoint paths cannot be dependent on one another. For a pure in-order execution model, the compiler would use the disjointness information and schedule the code accordingly. Two definitions of the same register guarded by disjoint predicates could safely be scheduled in the same cycle because only one definition would be guarded by true and ultimately be written. A definition and use on disjoint paths could also be scheduled in the same cycle for the same reason. PQS has been included in the later phases of the Intel IA-64 Compiler Code Generator [7] in the form of a relational database from which information on predicate disjointness, dominance, post-dominance and predicate promotion [14] can be obtained.

In [9], we presented the need for complete path analysis for predicated regions. This work extended the PQS research by maintaining information not only on predicate disjointness, but on the predicate predecessor/successor relationships that re-create a path through a predicated region. We presented the idea of Full Path Predicates (FPPs) to create predicates that represent a path through the predicated region. This allows statements to be predicated on the path that was taken to reach the statement, rather than only with a particular basic block as in predicates created by if-conversion. This allowed greater flexibility in instruction scheduling, speculation, and control height reduction.

August et. al. [6] and Sias et. al [19] examined using Binary Decision Diagrams (BDD) to provide accurate and efficient predicated execution and analysis. They used BDDs to completely represent predicate relationships. Unlike PQS, the BBDs have no loss due to their representation. Any form of query can be made once expressed as a Boolean expression and tested as a tautology. They found this extra power essential when dealing with predicates created in optimizations following if-conversion.

PQS, BBDs, and the analysis designed for creating FPPs were compile-time solutions to filling the need for specialized dependency analysis of predicated code for instruction scheduling. However, when the instructions are executed in hardware, the hardware must now deal with resolving these multiple definitions. The motivation for research presented in this paper is to design a way to perform the analysis described above in the context of a real-time hardware environment, to eliminate the stalls described in Section 2.1.

```
      add r5=r2,r4
      cmp P2,P3 =r5,0
(P2) mov r5 =r7
(P3) add r6=(r5or r5 ),3
(P3) cmp P4,P5 = r6,0
(P4) mov r5  =3
(P4) mov r6 =-1
(P5) mov r6  =0
      mult r9=(r5or r5 or r5  ),(r6 or r6  or r6  )
      cmp P5,P6 = r4, (r5 or r5  or r5  )
```

Figure 4: If Converted showing renamed definitions of same register

3.2 Hardware Solutions for Dealing with Multiple Path Definitions

Wang et.al. [20] recognized that multiple definitions would be a problem in the renaming stage of an out-of-order implementation of an architecture supporting predicated code. The renaming stage is used to give each definition of an architectural register a unique physical name (removing WAW and WAR dependencies). In the presence of 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.

Figure 4 provides a renamed version of the code in Figure 2 and illustrates the problem that can occur in the renaming stage. The mappings of the first few statements are unambiguous. The physical register `r5` defined by the first `add` can be mapped to the first use in the `cmp`. However, it is unclear which renamed version of `r5` should be mapped to the use of `r5` in the second `add`. It is even more ambiguous which definition will reach the use in the `mult`. This ambiguity cannot be removed until the values of predicate registers `P2` and `P4` are known.

In an effort to remove as many unnecessary stalls as possible, Wang et. al. proposed the use of the select-μop instruction for an IA64 out-of-order execution model. The new select-μop instruction was based on the phi-node used by static-single-assignment (SSA) [10]. It allows the resolution of multiple definitions to be postponed to later stages of the pipeline, providing more chance that a stall would not have to occur. To form the select-μop instructions, the possible definitions need to be kept track of. To this end they presented an augmented *Register Alias Table* (RAT), and use this to create the select-op instructions. The RAT used for this optimization is shown in Figure 5. Each set represents the current logical definitions for a given register. Each block contains the renamed definition and the Guarding Predicate under which it was defined. The most recent definition guarded by a true predicate would be the correct

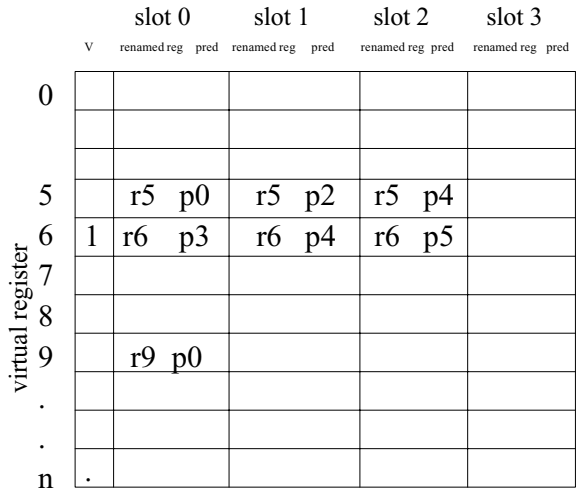


Figure 5: Augmented Register Alias Table used to implement the select-μop optimization for out-of-order processors supporting predicated execution.

definition. Once this definition was determined, any further dependencies left to be reconciled could be eliminated from consideration.

The entries are ordered in the table with the most recent definition in the highest numbered slot. Each set has four entries. If there is no available slot when a new definition must be entered, a select-μop is created, replacing all four entries. A select-μop is also created and inserted into the instruction stream when any use of a register with multiple definitions is encountered in the rename stage. For example, the first few instructions from Figure 4 with the select-μop included would be:

```
    add r5=r2,r4
    cmp P2,P3=r5,0
(P2)mov r5'=r7
    select r5=r5,r5'
(P3)add r6=r5,3
```

This removes any ambiguity as to which renamed version of `r5` should be mapped to the use in the final `add` instruction, avoiding the need to stall in the rename stage. A stall might be necessary if the guarding predicates of the definitions of `r5` are not determined by the time the select-μop executes.

The select-μop is not designed to be helpful for in-order processors. It is an optimization for the renaming stage, which is not a part of the in-order pipeline. In addition, it inserts another instruction and layer of dependency into the instruction stream. We use a modified version of the RAT for Disjoint Path Analysis architecture described in the next section. We do not store renamed physical registers in the RAT, since we are concentrating on an in-order VLIW architecture.

4. DISJOINT PATH ANALYSIS ARCHITECTURE

In this section, we describe the Disjoint Path Analysis Architecture that allows the same predicate sensitive path analysis done in the compiler to be accomplished in the hardware. To support this analysis, we add the Path Information Table (PIT) and the Last Definition Table (LDT) to the Itanium Implementation. In addition, we replace the register status table (the mechanism for determining if there is an outstanding write of each of the logical registers) with an extended Register Alias Table (RAT). These tables can be seen in Figures 7, 8 and 9 in various stages as we progress through processing the code in Figure 3. Register Alias Tables are common to out-of-order processors to facilitate renaming. We use a modified version to maintain information about definitions that define the same logical register from the different paths combined during if-conversion. These registers are not renamed, as in the out-of-order use of the RAT [20], since we are modeling an in-order VLIW processor. The other structures are unique to the Disjoint Path Analysis Architecture. The PIT is used to maintain disjointness information about predicates that guard register definitions and uses. The LDT is used to provide information to the RAT about the latest reference in the PIT to a particular predicate definition.

The rest of this section describes these structures in more detail. In particular, we will discuss how these structures are utilized and updated in the Disjoint Path Analysis Architecture to provide information critical to eliminating non-essential register dependencies.

4.1 Register Alias Table

In an EPIC architecture, there can be multiple possible register definitions for a given operand as described previously. As shown in Section 2, extraneous definitions could cause unnecessary stalls in the pipeline.

To facilitate the process of determining the correct definition, an extended *Register Alias Table (RAT)* is used, adapted for our purposes from [20]. The RAT implementation is used to maintain a list of the possible definitions for the use of a logical register. Since we are concentrating on an in-order model, the multiple definitions are not due to renaming. Instead, they are the result of the same logical register definition along different disjoint paths merged together through if-conversion. Consequently, each entry in a set for a logical non-predicate register contains not a new physical register, but a reference to the instruction that created the particular definition.

The second part of the slot entry is not just the predicate on which the definition was guarded, but a reference to the location in our Path Information Table where disjointness information associated with the definition's guarding predicate can be found. The Path Information Table entry comes from the LDT. If the entry in the LDT corresponding to the guarding predicate of the current instruction is valid, this information is recorded in the RAT. If the entry is invalid, no reference to disjointness information is made. This will be discussed further in Section 4.2.

The RAT is updated in the decode stage for each register definition with one exception. If there is not an empty slot when a new register definition is encountered, the defining instruction stalls the whole front-end of the processor until a slot has been vacated. We modeled the RAT using various numbers of slots ranging from 4 to 16. With 16 slots the pipeline never had to stall due to lack of available slots. With four slots, we did have to stall occasionally, but the effects on the IPC were relatively small as we will show in

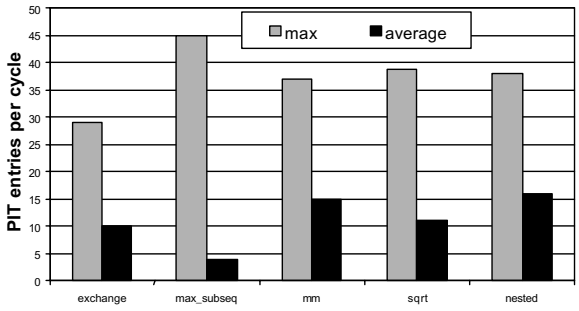


Figure 6: Maximum and average number of PIT entries required per cycle to support references in the RAT.

Section 6.

4.2 Predicate Information Table and Last Definition Table

The Predicate Information Table (PIT) is updated to maintain information on disjointness between predicates. The table is then used to answer the question:

- **Can two given instructions possibly be on the same path?**

This is critical information because a dependence cannot exist between two instructions that are not on the same path. If we can answer this question about the second `add` instruction and the first `mov` instruction in Figure 2, we can know that the use of `r5` in the `add` cannot be dependent on the definition of `r5` in the `mov`.

As in the example in Figure 7 depicts, the PIT is a matrix NxN representing the last N definitions of predicate registers. A given logical predicate definition can be represented multiple times in the table. We refer to these definitions as predicate definition *instances* because multiple vectors may reflect information about the same logical predicate register, only a different instance of its definition. For example, Figure 2 shows `P5` being defined in two different places in the code. Each of these definitions will have different disjointness information associated with them and therefore must be represented separately. The rows in the PIT represent the predicate register instances with which the given predicate definition (represented by the column) is disjoint.

Each of these definition instances must remain *live* until it is no longer possible that an instruction guarded by it remains in the pipeline. Consider the following definitions:

```
(1)      cmp P4,P5 = r8,r5
(2) (P4) mov r7=r5
(3) (P5) cmp P9,P4 = r9,r5
(4) (P4) add r6=r7,3
```

When the dependencies for the final `add` are calculated, it is important to have the disjointness information for the prior definition of `r7` available. The `add` and the `mov` are guarded by the same logical predicate register, but are not necessarily on the same path. To maintain correct disjointness information for the `mov`'s definition, we cannot allow the latest definition of `P4` to replace the previous definition in the PIT.

The first PIT definition for `P4` can be freed once the second definition of `P4` in our example is committed. Since we are executing instructions in order, at this point we are guaranteed that there are no instructions guarded by this definition left in the pipeline. In addition, all of the predicates represented in the PIT are unconditional predicate defines, so they are guaranteed to define their two predicates even if their guarding predicate is false. For example, in statement 3, `P9` and `P4` will be defined even if `P5` is false.

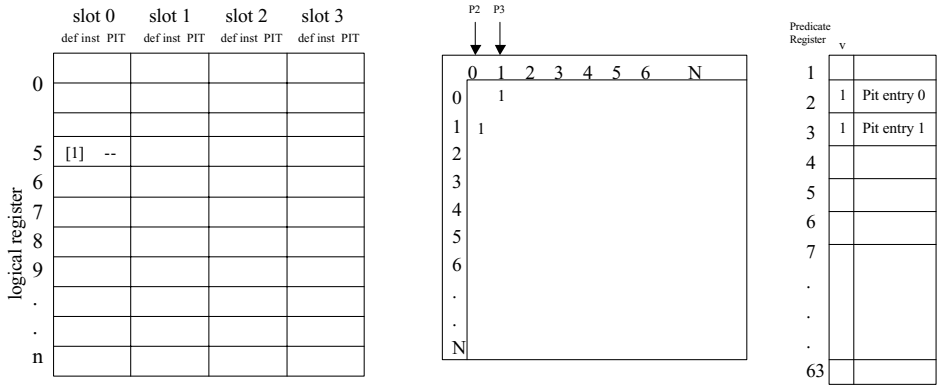
A disjoint representation in the PIT, means that those two predicates are guaranteed to be disjoint. If two predicates are not represented as disjoint in the PIT, then no disjoint information is know about those two predicates. They either may or may not be disjoint. The PIT only represents disjointness information between two predicates that it can guarantee to be disjoint.

Any PIT column entry can be allocated to a given predicate definition during execution. Each predicate definition that occurs during execution is allocated the first free PIT vector. We maintain a queue of pointers to free pit vectors. The deallocation of PIT entries is handled by the LDT and described below. If there is not a free PIT entry (the free list is empty), no disjointness information will be recorded about the predicate definition instance. We used a 45x45 matrix in this paper and never encountered a lack of available PIT entries. Figure 6 shows the maximum and average PIT entries that the RAT referenced in a given cycle. The maximum number of entries required ranged from 29-45 for the benchmarks we tested. The average entries used was significantly less.

The matrix is initialized so that every entry (corresponding to a column) is a bit vector set to 0. To determine which predicate definition instances are disjoint from another predicate instance `x`, the column of `x` is read. This produces a bit vector representing the disjointness of predicate `x` in relationship to all of the other predicate definitions represented in the PIT. A bit set in the `n`th location of the vector indicates that Predicate definition instance `n` is disjoint from instance `x`.

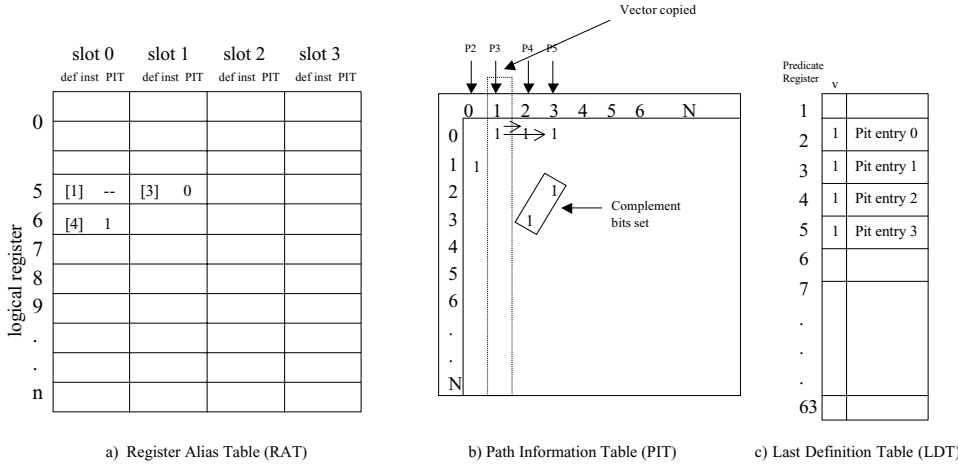
The disjointness information is accumulated in the PIT from 2 sources. First, if the predicate defining statement is guarded by a predicate, the disjointness information is inherited from its guarding predicate. In Figure 2 `P3` guards the definitions of `P4` and `P5`, so `P4` and `P5` are successors of `P3`. Both `P4` and `P5` inherit the disjointness information from `P3` in the PIT. If the predicate defining statement is not guarded, or guarded by P0 (the constant *true*), the predicate is initialized to be not disjoint from any predicate definitions. In our example, the definitions of `P6` and `P5` (second instance) are unguarded, so they do not inherit disjointness information. Second, disjointness information is added to the PIT stating that the newly defined predicates (`P5` and `P6`) are disjoint.

The Last Definition Table is shown in Figures 7, 8 and 9 at various stages as entries are made. It contains two entries for each logical predicate register. One entry is a pointer to the Column of the PIT representing the last definition instance of a given predicate register. For example, LDT[4] will contain a pointer to the last unconditional definition of `P4`, and LDT[5] to the last definition of `P5`. The other field in the LDT entry indicates if the last definition of the logical register contains a valid PIT entry. It may not be valid if either of the following are true:



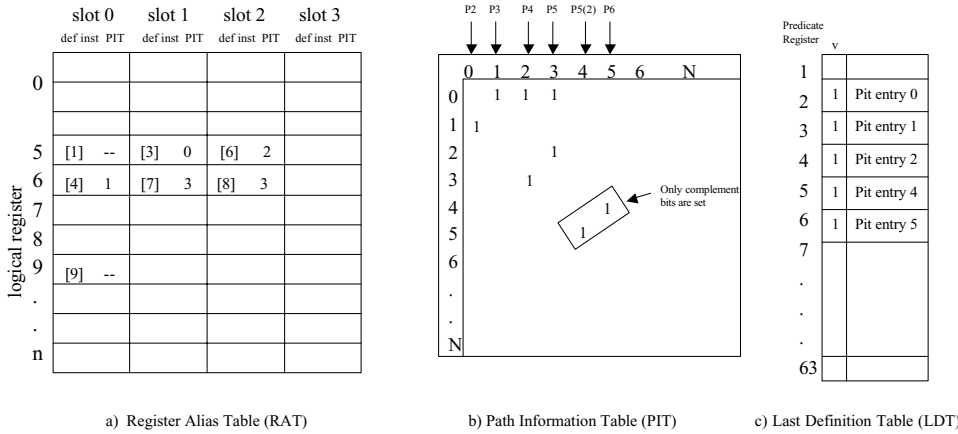
a) Register Alias Table (RAT) b) Path Information Table (PIT) c) Last Definition Table (LDT)

Figure 7: Three tables shown after first `cmp` statement is processed. One definition of `r5` has been entered into the RAT. As is it unguarded, there is no PIT entry associated with it. In the PIT, the bits are set at the intersections of locations 0 and 1 indicating that predicate registers `P2` and `P3` are disjoint. The LDT indicates that the current definition of predicate register 2 is found in PIT entry 0, and predicate register 3 at PIT entry 1.



a) Register Alias Table (RAT) b) Path Information Table (PIT) c) Last Definition Table (LDT)

Figure 8: Three tables after second `cmp` statement is processed. Two new definitions have been added to the RAT and the LDT. In the PIT, the definitions of `P4` and `P5` are guarded by `P3`, so the disjointness information for `P3` is inherited by `P4` and `P5`.



a) Register Alias Table (RAT) b) Path Information Table (PIT) c) Last Definition Table (LDT)

Figure 9: Three tables after third `cmp` statement is processed. A new definition of `P5` is made and given its own disjointness information. The old definition of `P5` in the LDT is replaced. Three new definitions are added to the RAT.

- There was not an available PIT vector when the latest definition was encountered.
- The last definition of the predicate was conditional. This could mean that it was a type of `or` or `and` predicate definition [3]. We do not keep disjoint information for these definitions in the PIT. For the benchmarks we tested, these types of definitions comprised an average of 0.4% of the predicate definitions.

When a predicate definition is added to the LDT, the LDT entry to be updated contains the last definitions PIT entry. This last (old) PIT entry is associated with the `cmp` instruction that is redefining that predicate. When this `cmp` instruction commits, it causes the deallocation of the PIT entry associated with the prior definition. When the PIT entry is deallocated, an entry is made in the PIT free list with the pointer to the deallocated vector. This scheme is similar to register deallocation for out-of-order processors.

The LDT information is used by the RAT to find the PIT vector of the guarding predicate of a register definition. This will be explained in more detail in Section 4.5.

On a branch prediction, our implementation will checkpoint both the PIT and the LDT. If a misprediction is determined, the PIT and LDT will be restored to their pre-branch condition. It is not necessary to checkpoint the RAT, as all instructions can continue through the pipeline, committing pre-branch definitions and squashing mis-predicted ones. If we wish to avoid the cost of checkpointing, an alternative is to clear the LDT and PIT on a mis-prediction and assume no disjointness information is available on recent predicate definitions.

4.3 Using the PIT and RAT to Determine Actual Dependencies

The PIT is accessed at most once per instruction when dependencies are being set. The RAT will be accessed N times where N is the number of operands in the current instruction. These tables are used as follows:

- **Current instruction is Guarded.** When processing an instruction, the vector associated with the guarding predicate of that instruction is read from the PIT. To determine which PIT entry to read, the PIT entry corresponding with the last definition of the predicate we are guarding is looked up in the LDT. Therefore, the LDT and PIT vector read for processing an instruction is done serially. For each operand in the current instruction, their corresponding RAT entries are read in parallel. These entries can be read in parallel with the LDT and PIT lookups. Each RAT entry contains contains a pointer to the dynamic instruction producing the value for that definition, along with a pointer to the PIT entry that represents that defining instructions guarding predicate. This PIT pointer is used to compute the disjointness of the RAT entry definition from the use for the instruction we are processing. The PIT pointer of each definition is looked up in the PIT vector of the current instruction's guarding predicate. If the bit for that PIT pointer is set in the vector, the definition associated with that index cannot create a dependency for the current instruction because the PIT pointer and the guarding predicate are disjoint. If not, or there is no PIT index in the RAT entry, a dependency *may* exist and the architecture will treat that definition as a potential input dependency.

  Figure 8 shows the PIT when the second `add` instruction in Figure 3 is considered. Entries have been made for definition instances of `P2` and `P3`. At this time the only definitions in the RAT for `r5` would be `r5[1]` and `r5[3]`. From PIT column 1 (the entry for `P3`, the guarding predicate of the `add`) we see that the bit for the location associated with `P2` (Row 0) is set. This means that the guarding predicate instance of definition `r5[3]` is disjoint from the instance of the guarding predicate of the `add`. Consequently `r5[3]` should not be considered as a possible definition, and will not set a dependence. However, `r5[1]` will set a dependence since it is unguarded and is not disjoint.

- **Current instruction not guarded or Guarded by P0.** The PIT is not accessed. Each possible definition for an operand in the RAT is a possible dependency to the use, and we cannot narrow the dependencies down with our disjointness information.

4.4 Updating the RAT

The RAT is updated in the decode and writeback stages except as mentioned earlier. When a defining instruction is processed, the register defined is entered into the RAT for the set corresponding to the logical register. It will be assigned a new available slot without replacing another with the following exceptions:

- An entire set in the RAT will be cleared for a logical register if the next entry to be made is un-guarded, or guarded by `P0` (the constant TRUE).
- A single slot will be replaced when a logical register is re-defined guarded by the same predicate instance (same PIT entry).

Two items of information are placed into the RAT entry. The first is a pointer to the instruction making the definition. The second is a pointer to the guarding predicate's disjointness information (if any) found in the PIT. The later piece of information is available from LDT if the valid bit indicates that the disjointness information is current.

RAT entries are removed during writeback when the defining instruction is committed or invalidated due to a guarding predicate with the value of false.

4.5 Updating the PIT

After the operands are processed for an instruction, the definitions are examined. If we have an unconditional `cmp` statement (used henceforth to represent all predicate defining statements), the instruction will update the PIT for the predicates defined. The next two free entries in the PIT will be cleared and allocated to the two new predicate definitions. The LDT will be updated to reflect the new most current definitions of the predicate registers defined. If we have a conditional predicate definition, the LDT valid bit will be set to invalid and no pit entries will be made. Two writes are performed into the PIT vectors, according to the rules below:

- **Inherit Disjointness Information**

– **A Guarded cmp**. If the `cmp` is guarded, the PIT entry corresponding to the guarding predicate is used to initialize the two vector PIT entries for the two new definitions. In doing this, we capture the inherited disjoint set information. Figure 8 shows the effect created when the second `cmp` statement is processed. The `cmp` statement was guarded, so the entries in the column of the guarding predicate instance of `P3` are copied into the the columns of the newly defined instances of `P4` and `P5`. The vector allocated to guarding predicate `P3` was determined from LDT[3]. The complement bits are set as described below.

– **An Unguarded cmp**. If the `cmp` is not guarded, or guarded by `P0` (the constant TRUE), the vectors of the newly defined predicates remain cleared. Figure 9 shows the PIT after the third `cmp` statement is encountered. Only the complement bits are set according to the next rule. Notice that the second instance of `P5` has different disjointness information than the first instance and that the entry in the LDT for `P5` pointer to the last definition.

- **Set Bit for Complement Predicate**. The two predicates defined in an unconditional `cmp` statement are always disjoint. We set the location of the complementary predicate in each of the vectors to indicate this.

  Figure 9 shows the complement bits set for the third `cmp`. `P5(2)` and `P6` are complementary predicates. `P5(2)` is allocated vector 4, so this location is set in vector 5 allocated to `P6`.

4.6 Predicates Defined False

Statements guarded on false predicates are by definition not on the executed path. Consequently, they never create dependencies and can be considered to be disjoint from every valid path. Their relationships to other invalid paths are inconsequential. The PIT can reflect this information. When a false predicate definition is encountered, the complete associated row and column in the PIT will be set.

5. METHODOLOGY

We created an EPIC simulator using the Itanium ISA derived from SimpleScalar [8] called IA64SimpleScalar. We extended the baseline SimpleScalar model to simulate the IA64 ISA in 5 areas. First, we added the ability to model in-order execution with detection and enforcement of false (WAW) dependencies. Second, we extended the simulator to support the IA64 predicated instruction set. One of the most significant changes in this area was the need to include the possibility of multiple definitions for the use of a register. Multiple definitions exist when the same register is defined along multiple paths that are joined into one through if-conversion. IA64 supports software pipelining, and we implemented the functionality of rotating registers along with specialized instructions that implicitly re-define predicate registers. Another important feature of the IA64 ISA is its support for control and data speculation [13]. To support this feature, we modeled the implementation of the ALAT, with its related operations and penalties. Finally, we added the ability to detect bundles and stop bits and appropriately issue instructions using this information.

Our simulator uses instruction traces instead of emulation. Our traces are generated on IA64 machines running Linux through the ptrace system interface [11]. This allows a parent program to single-step a spawned child. For each instruction in the trace, we record the information necessary to simulate the machine state. For all instructions, we record the instruction pointer, the current frame marker [2] and the predicate registers. In addition, we record the effective-address for memory operations and the previous function state for return. The data collected is written to a trace file. IA64SimpleScalar then reads in the trace file to simulate the program's execution.

IA64SimpleScalar decodes the traces using an opcode library containing a record for each IA64 instruction, and a library that interprets each instruction. This was built from the GNU opcode library that contains opcode masks to match the instruction, operand descriptions, mnemonic, and type classification. We enhanced this by adding a unique instruction identifier, quantity of register writers, and target Itanium functional unit.

In this paper we used a number of small benchmarks, chosen for structure which would produce predication. Two of the benchmarks we use are from the Trimaran System [1]. These include `mm` and `sqrt`. In addition, we included a program that completes an exchange sort, a program that computes the maximum subsequence found in a list of numbers and test program called `nested` created in an effort to find a program that contained more if-conversion. Table 1 provides a description of each benchmark, the number of instructions in the trace, the percent of if-converted instructions produced, and the initial IPC without the path optimization applied. All benchmarks were compiled using the Intel IA64 shrink wrapped C++ Compiler using the -O3 compilation option with profiling. Current production compilers produce minimal code guarded on predication. This is a result from having most of the predication turned off in the compiler due to the maturity level of the compiler and some predicate implementation issues in Itanium. We believe that future generations of EPIC processors and compilers will perform more predication, and this will increase the benefit one can expect from using our Disjoint Path Analysis architecture.

Table 2 shows the parameters used for the simulated microarchitecture modeled after the Itanium. The functional unit distribution includes 2 integer units, 2 memory units (able to execute some IALU instructions as well), 2 floating point units and 3 branch functional units. The latencies used varied by instruction, and were derived from the Itanium Processor Microarchitecture Reference [4]. The memory hierarchy is modeled after the Itanium. The L1 data and instruction caches are 4-way associative, sized at 16K with 32 byte blocks. The L2 cache is unified, with a capacity of 96K, 6-way set-associative with base latency of 6 cycles. Floating point loads bypass the L1 cache and incur an extra 3 cycle latency in the L2 cache (totaling 9). The L2 cache will allow misses on as many as 8 outstanding cache lines at once [4]. Loads made from an address to which a value was stored within the last 3 cycles will bypass the L1 cache, seeking its value from the L2 cache. The L3 cache is unified, located off chip in the Itanium implementation. The base latency is 21 cycles with floating point loads requiring 24.

benchmark	trace size	% if-conversion	% predication	io IPC	description
exchange	506969	10.2	20	.8083	Exchange sort routine
max_subseq	1421327	3.3	57	.7954	Finds maximum subsequence in a list
mm	937964	15	33	1.0373	matrix multiplication and summation
sqrt	560019	11.2	26	.8169	Newton-Raphson - saves partial results in array
nested	4635904	16.9	25	.9726	Nested loops and if-then-elses

Table 1: Presents description of benchmarks simulated including instruction count, percent of dynamic if-converted instructions and baseline IPCs for in-order and out-of-order execution.

L1 I Cache	16k 4-way set-associative, 32 byte blocks, 2 pipeline cycles
L1 D Cache	16k 4-way set-associative, 32 byte blocks, 2 cycle latency
Unified L2 Cache	96k 6-way set-associative, 64 byte blocks, 6 cycle latency
Unified L3 Cache	2Meg direct mapped, 64 byte blocks, 21 cycle latency
Memory Disambiguation	load/store queue, loads may execute when all prior store addresses are known
Functional Units	2-integer ALU, 2-load/store units, 2-FP units, 3-branch
DTLB, ITLB	4K byte pages, fully associative, 64 entries, 15 cycle latency
Branch Predictor	meta-chooser predictor that chooses between bimodal and 2-level gshare, each table has 4096 entries
BTB	4096 entries, 4-way set-associative

Table 2: Baseline Simulation Model created to correspond the the parameters set by the Intel Itanium.

Although the Itanium implements two levels of DTLB, we modeled the DTLB similar to the ITLB as fully associative, with 64 entries, 4k page size and a 15 cycle latency. Memory access is assumed to require 80 cycles. All models issue up to 6 instructions per cycle. Branch misprediction penalty is a minimum of 8 cycles.

6. DISJOINT PATH ANALYSIS RESULTS

We apply Disjoint Path Analysis Architecture to our Intel Itanium in-order model derived from [13, 3, 7, 18].

The IA64 compiler *bundles* instructions into groups of three. A template included with the bundle is used to describe to the hardware the combination of functional units required to execute the operations in the bundle. Stop bits are inserted in the instruction stream as part of the template to create *instruction groups*, or sets of instructions that are known to be independent of each other. Stop bits guarantee that the instructions issued together are independent of each other, but give no information on the relationships of these instructions to those outside of the group.

In the Itanium implementation, bundles are directed or *dispersed* to the functional units two at a time, subject to independence and resource constraints. The full two bundles (up to 6 instructions) are sent unless a functional unit is unavailable, or a stop bit is encountered indicating the end of an instruction group. If any of the instructions entering the functional units must stall because a data dependence is not yet satisfied, all of the instructions waiting to begin execution stall as well. This is essentially a pure in-order processor requiring a limited scoreboard to determine when the whole instruction group can start execution.

The scoreboarding mechanism is used to detect dependencies between instructions outside of instruction groups. It allows dependencies to be broken when either the producing or consuming instruction is found to be guarded by a predicate evaluated to false. Our Disjoint Path Analysis architecture focuses on eliminating these false dependencies allowing instructions to start executing a little earlier as described in Section 2.1.

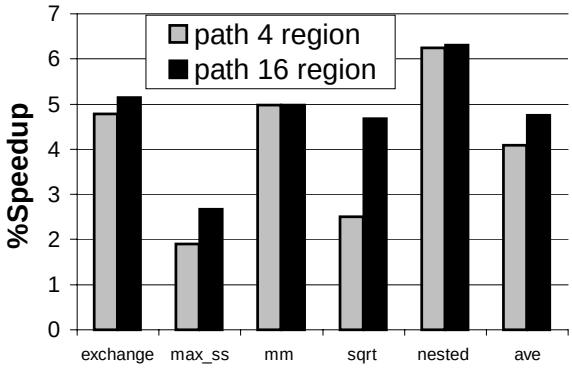


Figure 10: Disjoint Path Analysis results. Results show the percent speedup obtained inside of if-converted regions when 4 and 16 entries are used per RAT register definition.

6.1 Results

Figure 10 shows the percent speedup achieved using the Disjoint Path Analysis architecture. Results are shown for a RAT with 4 and 16 possible definitions per register. These results show the percent speedup over the percent of code executed in if-converted regions. Table 1 shows that only 3% to 17% of the executed code was in if-converted regions. The production compiler we used in this study was overly conservative in the if-conversion regions it formed, and future compilers most likely will create larger predicated regions. On average, results with a RAT of size 4 produced speedups of 4.1%, and results with a RAT of size 16 produced speedups of 4.8%

Figure 11 shows the percent speedup over the complete execution of the program if every use knew exactly which definition it should be dependent upon. This shows an upper bound of the potential benefit possible for knowing exact path information. This is important because if we had perfect knowledge of the values each predicate definition would ultimately assume, we could know exactly which dependen-

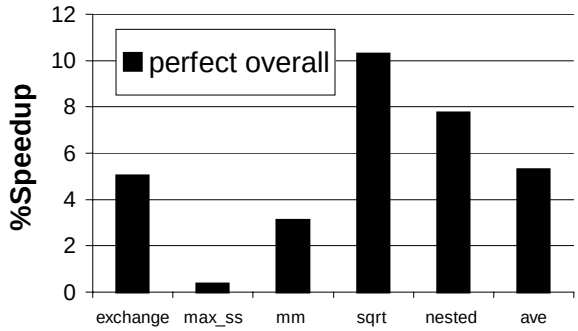


Figure 11: Percent speedup achievable over the complete execution of the program if every use knew perfectly which definition it depended upon.

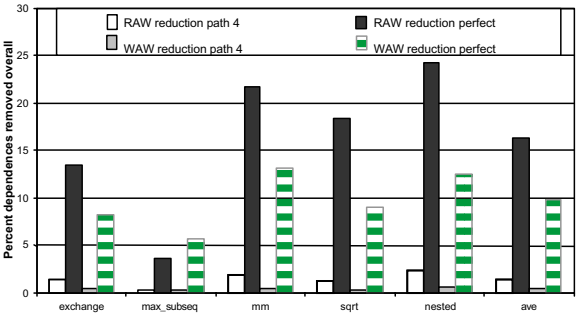


Figure 12: Percent of dependencies removed by Disjoint Path Analysis and perfect predicate information over the complete execution of the program. RAW represents Read After Write dependencies while WAW represents Write After Write dependencies.

cies to set. No dependencies would be set for instructions guarded by a predicate with the value of false, and the only definition to reach a use in a statement guarded by a true predicate would be the last definition guarded by a true predicate. The results show that having perfect information can achieve on average a speedup of 5.3% for the complete execution of the program.

Next, we compare the number of dependencies removed using our Disjoint Path Analysis Architecture with a four entry PIT set against those removed using perfect predicate information. As shown in Figure 12, the average number of Read After Write (RAW) dependencies removed was a little over 1% using the Disjoint Path Architecture while the number of Write After Write (WAW) dependencies removed was less than half of that. The average for having perfect predicate information was over 16% for RAW dependencies and almost 10% for WAW dependencies. This is because, using perfect predicate information, no dependencies were set between any producers or consumers guarded by false predicates. Note, Write After Read (WAR) dependencies are not an issue due to the nature of in-order processors.

The perfect predicate information knew of all false definitions. In contrast, Disjoint Path Analysis could only determine if a producer and consumer were on the same path. A dependence could initially be set between a producer and consumer both of which had false guarding predicates. Note that these dependencies will be broken by the scoreboard as soon as the producer or consumer is known to be guarded by false.

7. CONCLUSIONS

In this paper, we present an approach to dynamic path analysis used to expose erroneous data dependencies in the current Itanium architecture and in an out-of-order implementation of an IA64 architecture. We present the Disjoint Path Architecture that allows us to re-create predicate relationship information at runtime in hardware. This predicate relationship information provided by the addition of the Path Information Table and related logic allows us to answer the question "can a given definition be on the same path for a use?" If the answer to this question is no, then that definition is not considered as a possible dependence. Our results showed that the number of RAW dependencies set in if-converted regions could be reduced by 14% using Dynamic Path Analysis with the current Itanium Implementation. As a result, IPC could be increased up to 6% in these regions.

The benchmarks we used for this study were compiled using the Intel IA64 shrink wrapped C++ Compiler using the -O3 compilation option with profiling. Current production compilers produce minimal code guarded on predication. This is a result from having most of the predication turned off in the compiler due to the maturity level of the compiler and some predicate performance issues in Itanium. We believe that future generations of EPIC processors and compilers will perform more predication, and this will increase the benefit one can expect from using our Disjoint Path Analysis architecture.

We presented results using our Disjoint Path Analysis architecture for a pure in-order processor. A potential application of this technique would be for an IA64 implementation employing out-of-order execution. For this model, where instructions can begin execution as soon as their operands are ready, it is even more important to eliminate false dependencies, thus exposing additional instruction level parallelism. Comparing the use of this technique with the select-μop approach discussed in Section 3 is a topic of interest for future work.

We recognize that this dynamic dependence analysis adds additional hardware complexity. An alternative would be to generate this disjointness information in the compiler, augmenting the analysis that already exists. The disjointness information could be communicated to the hardware by a renaming phase late in the scheduling process. Separate names could be given to different instances of the same variables determined to be on different paths. The tradeoff, however, would be an increased data dependence depth. When two paths rejoin, a form of phi instruction would have to be inserted to determine which definition reaches a use. This is a topic of interest for future work as well.

Acknowledgments

We would like to thank the anonymous reviewers for providing useful comments on this paper. This work was funded in part by NSF grant No. 0073551, a grant from Intel Corporation, and an equipment grant from Hewlett Packard and Intel Corporation. We would like to thank Carole Dulong and Harpreet Chadha at Intel for their assistance bringing up the Intel IA64 Electron compiler.

8. REFERENCES

[1] Trimaran, An Infrastructure for Research in Instruction Level Parallelism, 1998. http://www.trimaran.org.

[2] IA-64 Application Instruction Set Architecture Guide, Revision 1.0, 1999.

[3] *Intel IA-64 Architecture Software Developer's Manual*. Intel, 2000.

[4] Itanium Processor Microarchitecture Reference:for Software Optimization., 2000. http://www.developer.intel.com/design/ia64/itanium.htm.

[5] J.R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, December 1994.

[6] D. August, J. Sias, J-M. Puiatti, S. Mahlke, D. Connors, K. Crozier, and W. Hwu. The program decision logic approach to predicated execution. In *Proceedings of the 26th International Symposium on Computer Architecture*, May 1999.

[7] 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, September 2000.

[8] 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.

[9] L. Carter, B. Simon, B. Calder, L. Carter, and J. Ferrante. Predicated static single assignment. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, October 1999.

[10] 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.

[11] S. Eranian and D. Mosberger. The Linux/IA64 Project: Kernel Design and Status Update. Technical Report HPL-2000-85, HP Labs, June 2000.

[12] D. M. Gillies, D. R. Ju, R. Johnson, and M. Schlansker. Global predicate analysis and its application to register allocation. In *Proceedings of the 29th Annual Intl. Symp. on Microarchitecture*, pages 114–125, December 1996.

[13] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir. Introducing the IA-64 Architecture. *IEEE Micro*, 20(5):12–22, September 2000.

[14] 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 *Proceedings of the 25th Annual Intl. Symp. on Microarchitecture*, pages 45–54, December 1992.

[15] J. C. H. Park and M. Schlansker. On Predicated Execution. Technical Report HPL-91-58, HP Labs, May 1991.

[16] D. Patterson and J. Hennessy. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, 1996.

[17] M. Schlansker and R. Johnson. Analysis techniques for predicated code. In *Proceedings of the 29th Annual Intl. Symp. on Microarchitecture*, pages 100–113, December 1996.

[18] H. Sharangpani and K. Arora. Itanium processor microarchitecture. In *IEEE MICRO*, pages 24–43, 2000.

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

[20] 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.