Improving the Performance of Object-Oriented Languages with Dynamic Predication of Indirect Jumps

José A. Joao‡   Onur Mutlu‡   Hyesoon Kim §   Rishi Agarwal†‡   Yale N. Patt**

*** High Performance Systems Group**
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas 78712-0240

‡Computer Architecture Group
Microsoft Research
Redmond, WA

§College of Computing
Georgia Institute of Technology
Atlanta, GA

†Department of Computer Science and Engineering
Indian Institute of Technology - Kanpur
Kanpur, India

TR-HPS-2008-001
January 2008

Improving the Performance of Object-Oriented Languages with Dynamic Predication of Indirect Jumps

José A. Joao*‡ Onur Mutlu‡ Hyesoon Kim § Rishi Agarwal†‡ Yale N. Patt*

* Department of ECE
Univ. of Texas at Austin
{joao, patt}@ece.utexas.edu

‡Comp. Architecture Group
Microsoft Research
onur@microsoft.com

§College of Computing
Georgia Inst. of Technology
hyesoon@cc.gatech.edu

†Department of CSE
IIT Kanpur
rishi@iitk.ac.in

Abstract

Indirect jump instructions are used to implement increasingly-common programming constructs such as virtual function calls, switch-case statements, jump tables, and interface calls. The performance impact of indirect jumps is likely to increase because indirect jumps with multiple targets are difficult to predict even with specialized hardware.

This paper proposes a new way of handling hard-to-predict indirect jumps: dynamically predicating them. The compiler (static or dynamic) identifies indirect jumps that are suitable for predication along with their control-flow merge (CFM) points. The hardware predicates the instructions between different targets of the jump and its CFM point if the jump turns out to be hard-to-predict at run time. If the jump would actually have been mispredicted, its dynamic predication eliminates a pipeline flush, thereby improving performance.

Our evaluations show that Dynamic Indirect jump Predication (DIP) *improves the performance of a set of object-oriented applications including the Java DaCapo benchmark suite by 37.8% compared to a commonly-used branch target buffer based predictor, while also reducing energy consumption by 24.8%. We compare DIP to three previously proposed indirect jump predictors and find that it provides the best performance and energy-efficiency.*

1. Introduction

Indirect jumps are becoming more common as an increasing number of programs is written in object-oriented languages such as Java, C#, and C++. To support polymorphism [8], these languages include virtual function calls that are implemented using indirect jump instructions in the instruction set architecture (ISA). Previous research has shown that modern object-oriented languages result in significantly more indirect jumps than traditional languages [7]. In addition to virtual function calls, indirect jumps are commonly used in the implementation of programming language constructs such as switch-case statements, jump tables, and interface calls [2].

Unfortunately, current pipelined processors are not good at predicting the target address of an indirect jump if multiple different targets are exercised at runtime. Such hard-to-predict indirect jumps not only limit processor performance and cause wasted energy consumption but also contribute significantly to the performance difference between traditional and object-oriented languages [44]. The goal of this paper is to develop new architectural support to improve the performance of programming language constructs implemented using indirect jumps.

Figure 1 demonstrates the problem of indirect jumps in object-oriented Java (DaCapo [5]) and C++ applications. This figure shows the indirect and conditional jump mispredictions per 1000 retired instructions (MPKI) on a state-of-the-art Intel Core2 Duo 6600 [22] processor. The data is collected with hardware performance counters using

VTune [23]. Note that the Intel Core2 Duo processor includes a specialized indirect jump predictor [16]. Despite specialized hardware to predict indirect jump targets, 41% of all jump mispredictions in the examined applications are due to indirect jumps. Hence, hard-to-predict indirect jumps cause a large fraction of all mispredictions in object-oriented Java and C++ applications. Therefore, more sophisticated architectural support than "target prediction" is needed to reduce the negative impact of indirect jump mispredictions on performance of object-oriented applications.

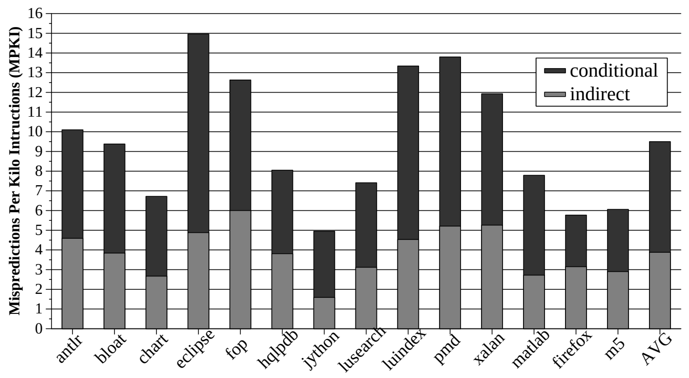

Figure 1. Indirect and conditional jump mispredictions in object-oriented Java and C++ applications run using the Windows Vista operating system on an Intel Core2 Duo 6600

Basic Idea: We propose a new way of handling hard-to-predict indirect jumps: *dynamically predicating them*. By dynamically predicating an indirect jump, the processor increases the probability of the correct target path of the jump to be fetched. Our technique stems from the observation that program control-flow paths starting from different targets of some indirect jump instructions usually merge at some point in the program, which we call the control-flow merge (CFM) point. The static or dynamic compiler[1] identifies such indirect jump instructions along with their CFM points and conveys them to the hardware through modifications in the instruction set architecture. When the hardware fetches such a jump, it estimates whether or not the jump is hard to predict using a confidence estimator [25]. If the jump is hard to predict, the processor predicates the instructions between N targets of the indirect jump and the CFM point. We evaluate performance/complexity for different N, and find N=2 is the best trade-off. When the processor reaches the CFM point on all N different target paths, it inserts select-μops to reconcile the data values produced on each path and continues execution on the control-independent path. When the indirect jump is resolved, the processor stops dynamic predication and turns the instructions that correspond to the incorrect target address(es) into NOPs as their predicate values are false. The instructions -if any- that correspond to the correct target address commit their results. As such, if the jump would actually have been mispredicted, its dynamic predication eliminates a full pipeline flush, thereby improving performance.

[1]In the rest of the paper, we use the term "compiler" to refer to either a static or dynamic compiler. Our scheme can be used in conjunction with both types of compilers.

Our experimental evaluation shows that *Dynamic Indirect jump Predication (DIP)* improves the performance of a set of indirect-jump-intensive object-oriented Java and C++ applications by 37.8% over a commonly-used branch target buffer (BTB) based indirect jump predictor, which is employed by most current processors. We compare DIP to three previously proposed indirect jump predictors [9, 13, 30] and find that it provides significantly better performance than all of them. Our results also show that DIP provides the largest improvements in energy-efficiency and energy-delay product.

We analyze the hardware cost and complexity of DIP and show that if dynamic predication is already implemented to reduce the misprediction penalty due to conditional branches [31], DIP requires little extra hardware. Hence, DIP can be a promising, energy-efficient way to reduce the performance penalty of indirect jumps without requiring large specialized hardware structures for predicting indirect jumps.

Contributions. We make the following contributions:

1. We provide a new architectural approach to support indirect jumps, an important performance limiter in object-oriented applications. To our knowledge, DIP is the first mechanism that enables the predication of indirect jumps.

2. We extensively evaluate DIP in comparison to several previously-proposed indirect jump prediction schemes and show that DIP provides the highest performance and energy improvements in modern object-oriented applications written in Java and C++. Even when used in conjunction with sophisticated predictors, DIP significantly improves performance and energy-efficiency.

3. We show that DIP can be implemented with little extra hardware if dynamic predication is already implemented to reduce the misprediction penalty due to conditional branches. Hence, we propose using dynamic predication as a general framework for reducing the performance penalty due to unpredictability in program control-flow (be it due to conditional branches or indirect jumps).

2. Background on Dynamic Predication of Conditional Branches

Compiler-based predication [1] has traditionally been used to eliminate conditional branches (hence conditional branch mispredictions) by converting control dependencies to data dependencies, but it is not used for indirect jumps. Dynamic predication was first proposed to eliminate the misprediction penalty due to simple hammock branches [34] and later extended to handle a large set of complex control-flow graphs [31]. Dynamic predication has advantages over static predication because (1) it does not require significant changes to the instruction set architecture, such as predicated instructions and architectural predicate registers, (2) it can adapt to dynamic changes in branch behavior, and (3) it is applicable to a much wider range of control-flow graphs and therefore provides higher performance [31]. Unfortunately, none of these previous static or dynamic predication approaches were applicable to indirect jumps.

We first briefly review the previous dynamic predication mechanisms proposed for conditional branches [34, 31] to provide sufficient background and the terminology used in this paper.

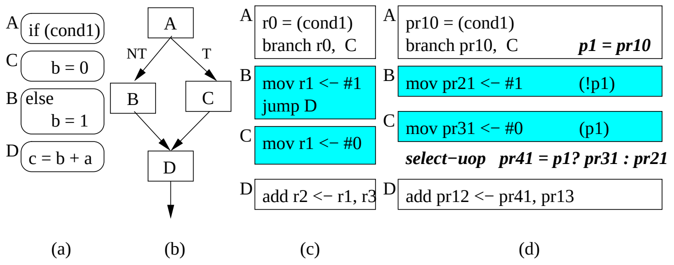

Figure 2. Dynamic predication of a conditional branch: (a) source code (b) CFG (c) assembly code (d) dynamically predicated instructions after register renaming (pr: physical register)

Figure 2 shows the control-flow graph (CFG) of a conditional branch and the dynamically predicated instructions. The candidate branches for dynamic predication are identified at runtime or marked by the compiler. When the processor fetches a candidate branch, it estimates whether or not the branch is hard to predict using a branch confidence estimator [25]. If the branch prediction has low confidence, the processor generates a predicate using the branch condition and enters *dynamic predication mode (dpred-mode)*. In this mode, the processor fetches both paths after the candidate branch and dynamically predicates the instructions with the corresponding predicate id. On each path, the processor follows the outcomes of the branch predictor. When the processor reaches a *control-flow merge (CFM) point* on both paths, it inserts c-moves [29] or select-μops [43], similar to the ϕ-functions in the static single-assignment (SSA) form [10], to reconcile the register data values produced on either side of the branch and continues fetching from a single path. The processor exits dpred-mode either when it reaches a CFM point on both paths of the branch or when the branch is resolved. When the branch is resolved, the predicate value is also resolved. Instructions on the wrong path (i.e. predicated-FALSE instructions) become NOPs, and they do not update the architectural state. If the candidate branch is actually mispredicted, the processor does not need to flush its pipeline and is able to make useful progress on the correct path, which provides improved performance.

3. Dynamic Predication of Indirect Jumps (DIP)

Traditionally, only conditional branches can be predicated because predication assumes that there are exactly two possible next instructions after a branch. This assumption does not hold for indirect jumps. Figure 3a shows an example virtual function call in the C++ language that is implemented as an indirect call (`s->area()`). Depending on the actual runtime type of the object pointed to by `s`, the corresponding overridden version of the `area` function will be called. There can be many different derived classes that override the function call and thus many different targets of the call. Even though there could be many different targets, usually only a few of them are concurrently used in each phase of the program. If the calls for different targets are interleaved in a complex way, it is usually difficult to predict *exactly the correct target* of each instance of the call using existing indirect jump predictors. In contrast, we found that it is much easier to estimate the two (or three) *most likely targets*, i.e. a small set of targets that includes the correct target with a high probability.

In DIP, if an indirect jump is found to be difficult to predict, the processor estimates the *most likely targets*. Using dynamic predication, the processor fetches and executes from these *most likely targets* until the dynamically-predicated paths eventually merge at the instruction after the call, when the function returns (as shown in Figure 3b,c). If one of the predicated targets is correct, the processor avoids a pipeline flush. The performance benefit of dynamically predicating the indirect jump can increase significantly if the control flow merging point is close to the indirect jump (i.e., if the body of the function is small), so that the overhead of fetching the extra path(s) is not high. Figure 3b,c illustrates conceptually the dynamic predication process for the indirect call in Figure 3a, assuming that `circle->area()` and `rectangle->area()` are the most likely targets for an instance of the `s->area()` call.

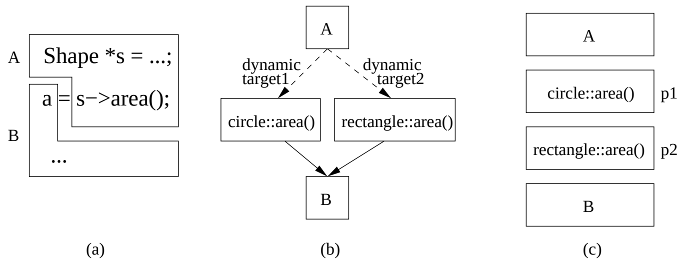

Figure 3. Dynamic predication of an indirect call: (a) source code (b) CFG (c) predicated code

Our approach is inspired by the dynamic predication of conditional branches. However, there are two fundamental differences between the dynamic predication of conditional branches and indirect jumps:

1. There are exactly two possible paths after a conditional branch. In contrast, the number of possible paths after an indirect jump depends on the number of possible targets, which can be very large. For example, an indirect call in the Java DaCapo benchmark *eclipse* exercises 101 dynamic targets. Predicating a larger number of target paths increases the likelihood that the correct path will be in the pipeline when the jump is resolved, but it also requires more complex hardware and increases the amount of wasted work due to predication since at most one path is correct. Therefore, one important question is *how to identify how many and which targets of a jump should be predicated*.

2. The target of a conditional branch is always available at compile time. On the other hand, all targets of an indirect jump may not be available at compile-time due to techniques like dynamic linking and dynamic class loading. Hence,

a static compiler might not be able to convey to hardware which targets of an indirect jump can profit from dynamic predication. Another important question, therefore, is *who (the compiler -static or dynamic- or the hardware) should determine the targets that should be dynamically predicated.* We explore both options: the compiler can determine the targets to be predicated via profiling or the hardware can determine them at runtime. Note that the latter option can adapt to runtime changes in frequently-executed targets of an indirect jump at the expense of higher hardware cost.

In this paper we explore answers to these questions and propose an effective and cost-efficient implementation of DIP.

4. Why does DIP work?

We first examine code examples from Java applications to provide insights into why DIP can improve performance.

4.1. Virtual Function Call Example

Figure 4 shows a virtual function call in `fop`, an output-independent print formatter Java application included in the DaCapo suite. The function `computeValue` is originally defined in the class `Length`, and is overridden in the derived classes `LinearCombinationLength`, `MixedLength` and `PercentLength`. This polymorphic function is called from a single call site 32% of the time by objects of class `Length`, 34% of the time by objects of class `LinearCombinationLength`, and 34% of the time by objects of class `PercentLength`. The benchmark goes through two program phases. Only the first target is used at the beginning of the program, and therefore the call is easy to predict. In the second phase the targets from `LinearCombinationLength` and `PercentLength` are interleaved in a difficult to predict way. Dynamically predicating these two targets when the indirect call becomes hard to predict can eliminate most target mispredictions at the cost of executing useless instructions on one path. Since the bodies of the functions are small, the number of wasted instructions with dynamic predication is smaller than the number of wasted instructions on a pipeline flush due to a misprediction.

```
public int mvalue() { // in Length class
  if (!bIsComputed)
      computeValue();     // call site
  return millipoints;
}

protected void computeValue() {
 // in LinearCombinationLength class, short computation...
 setComputedValue(result);
}

protected void computeValue() { // in MixedLength class
  // short computation...
  setComputedValue(computedValue, bAllComputed);
}

protected void computeValue() { // in PercentLength class
  setComputedValue((int)(factor *
    (double)lbase.getBaseLength()));
}
```

Figure 4. A suitable indirect jump example from `fop`

4.2. Switch-Case Statement Example

Figure 5 shows a switch statement in the function `jjStopStringLiteralDfa_0` of the class `JavaParserTokenManager` from the DaCapo benchmark `pmd`. This class parses input tokens by implementing a deterministic finite automaton. Even though the switch statement has 11 cases, cases 0, 1 and 2 are executed for 59%, 25%, and 12% of the dynamic instances, respectively. The other 8 cases account for only 4% of the dynamic instances. The control flow reconverges after the switch statement. Dynamically predicating the first three target paths when the indirect jump is seen would eliminate almost all mispredictions at the cost of executing useless instructions. Note, however, that the number of instructions is relatively small (fewer than 30) in each target path, so the amount of wasted work would be small compared to the amount of wasted work on a full pipeline/window flush due to a misprediction.

```
switch (pos) { // indirect jump
   case 0: // target 1
      if ((active1 & 0x40000000040000L) != 0L)
         r = 4;
      else if (...) ...
         r = 28;
      else
        r = -1;
     break;
   case 1: // target 2
      // code similar to case 0 (setting r on every path)
   case 2: // target 3
      // code similar to case 0 (setting r on every path)
   // ... 8 other seldom executed cases
}
```

Figure 5. A suitable indirect jump example from `pmd`

5. Mechanism and Implementation

There are two critical issues in implementing DIP: (1) determining which indirect jumps are candidates for dynamic predication, (2) determining which targets of a candidate indirect jump should be predicated. This section first explains how our mechanism addresses these issues. Then, we describe the required hardware support, analyze its complexity, and explain the support required from the ISA.

5.1. Indirect Jump and CFM Point Selection

The compiler selects indirect jump candidates for dynamic predication using control-flow analysis and profiling. Control-flow analysis finds the CFM point for each indirect jump. The CFM point for an indirect call is the instruction after the call. The CFM point for an indirect jump implementing a switch statement is usually the instruction after the statement. The compiler profiles the application to characterize the indirect jumps. Highly mispredicted indirect jumps are good candidates for DIP even if there is no CFM point common to all the targets or if the CFM point is so far from the jump that it is not reached until the indirect jump is resolved. In this case, DIP still could provide performance benefit because it executes two possible paths after the jump, one of which might be the correct path.

In other words, the benefit from DIP is similar to that of dual-path execution [17, 15] if a CFM point is not reached. For the experiments in this paper the compiler selects all indirect jumps that result in at least 0.1% of all jump mispredictions in the profiling run on the baseline processor.[2]

An indirect jump selected for dynamic predication is marked in the binary along with its CFM point. We call such a jump a *DIP-jump*.

5.1.1. Return CFM Points In some switch statements, one or more *cases* might end with a *return* instruction. For an indirect jump implementing such a switch statement, the first instruction after the statement might not be the CFM point. If all predicated paths after an indirect jump implementing a switch statement reach a return instruction that ends a *case*, the CFM point is actually the instruction executed after the return instruction. Unfortunately, the address of this CFM point is not known at code generation time because it depends on the caller position. We introduce a special type of CFM point called *return CFM* to handle this case. When a DIP-jump is marked as having a return CFM point, the processor does not look for a particular address to end dpred-mode, but for the execution of a return instruction at the same call depth as the DIP-jump. The processor ends dynamic predication mode when all the predicated paths reach return instructions.

5.2. Target Selection

DIP provides performance benefit only if the correct target of a jump is one of the predicated targets. Therefore, the choice of which targets to predicate is an important decision to make when dynamically predicating an indirect jump since only a few targets can be predicated. This choice can be made by the compiler or the hardware. We first describe how target selection can be done assuming two targets can be predicated. Section 5.2.3 describes the selection of more than two targets, assuming the hardware can support the predication of all of them.

5.2.1. Compiler-based Target Selection Even though an indirect jump can have many dynamically-exercised targets, we would expect the most frequently exercised targets to account for a significant fraction of the total dynamic jump instances and mispredictions [30, 27]. This assumption suggests using a simple mechanism for target selection: the compiler profiles the program with a representative input set, determines the most frequently executed targets for each DIP-jump, and annotates the executable binary with the target information. Even though this mechanism requires more ISA support to supply the targets with an indirect jump, it does not require extra hardware for target selection. However, our results show that dynamic target selection mechanisms that can adapt to runtime program behavior can be much more effective at the cost of extra hardware (see Section 7.4).

[2]We have experimented with several other profiling and selection heuristics based on compile-time cost-benefit analyses (including the ones described in [33, 27]), but we found that our simple selection heuristic provides the best performance.

5.2.2. Hardware-based (Dynamic) Target Selection The correct target of an indirect jump depends on the runtime behavior of the application. Changes in the runtime input set, phase behavior of the program, and the control-flow path leading to the indirect jump affect the correct target, which is actually the reason why some indirect jumps are hard to predict. As the compiler does not have access to such fine-grain dynamic information, it is difficult for the compiler to select a set of targets that includes the correct target when the jump is predicated. In contrast, hardware has access to dynamic program information and can adapt to rapid changes in the behavior of indirect jumps. We therefore develop a mechanism that selects targets based on runtime information collected in hardware.

We use a hardware table called *Target Selection Table (TST)* for dynamic target selection. The purpose of the TST is to track and provide the most frequently executed two targets for a given DIP-jump. A TST entry is associated with each DIP-jump. Conceptually, each entry in the TST contains M targets and M frequency counters. A frequency counter is associated with each target and keeps track of how many times the target was taken. When a fetched DIP-jump is estimated to be hard-to-predict (low-confidence), the processor accesses the TST entry for that jump and selects the two most frequently executed target addresses (i.e. the two target addresses with the highest frequency counters) in the entry.

The TST is structured as a 4-way set-associative cache with a least-recently-used (LRU) replacement policy. We evaluated different indexing functions for the TST: using the address (i.e., program counter) of the DIP-jump alone or the address of the DIP-jump XORed with the 16-bit global branch history register (GHR). We found that the latter indexing function provides more accurate target selection because the correct target of a jump depends on the control-flow path leading to the jump.

To reduce the storage requirements for the TST, we: (1) limit the number of targets to the maximum number of targets that can be predicated plus one; (2) implement the frequency counters as 2-bit saturating counters[3]; (3) limit the tag to 7 bits; (4) limit the size of the TST to 2K entries; (5) store the targets associated with a DIP-jump in the BTB (in different BTB entries), instead of storing them in the TST itself. The last optimization allows TST to become a low-cost indirection mechanism that stores only frequency-counters to retrieve the most frequently executed targets of a branch, which are stored in the BTB.

Operation of TST: When a fetched DIP-jump is estimated to be hard-to-predict, the target selection mechanism starts an iterative process to retrieve the most frequently used two targets from the BTB, one target per cycle.[4] Figure 6 shows the basic structure of the TST and the logic required to access the BTB based on the information obtained from

[3]To dynamically select 3 to 5 targets, we use 3-bit saturating frequency counters.
[4]The performance impact of the extra cycles spent to retrieve targets from the BTB is 2%, as we show in Section 7.7.

the TST.[5] Algorithm 1 describes the target selection process. In each iteration *iter*, the control logic finds the *position* of the next frequency counter in descending order. If there are 3 counters stored in the TST, *position* can take only the values 1, 2 or 3. The value used to access the BTB to retrieve a target is the same value used to index the TST XORed with a randomized constant value *hash_value*, which is specific to each *position*.[6] For example, if `f3` and `f1` are the highest frequency counters, the targets will be retrieved by accessing the BTB with `(PC xor GHR xor hash_value[3])` and `(PC xor GHR xor hash_value[1])` in consecutive cycles. The iterative selection process stops when it has the required number of targets to dynamically predicate the jump (*PRED_TARGETS*), or after trying to retrieve as many targets as can be stored for one TST entry (*MAX_TARGETS*). *PRED_TARGETS* is 2 and *MAX_TARGETS* is 3 for 2-target selection. If enough targets are selected, the processor enters dpred-mode.

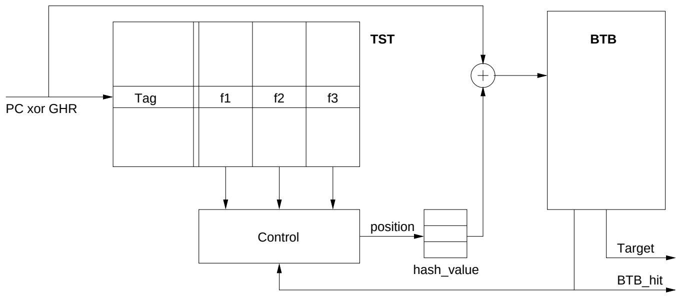

Figure 6. Target Selection Table (TST) used for selecting 2 targets to predicate. f1, f2, f3 denote the frequency counters for the three targets whose information is kept in TST.

Algorithm 1 TST target selection algorithm. Inputs: PC, GHR

```
$iter \leftarrow 1$
$num\_targets \leftarrow 0$
**while** $((iter \leq MAX\_TARGETS)\ and$
      $(num\_targets < PRED\_TARGETS))$ **do**
  $position \leftarrow position\_descending\_order(iter)$
  $target \leftarrow$ access_BTB($PC\ xor\ GHR\ xor\ hash\_value[position]$)
  **if** $(BTB\_hit)$ **then**
    $next\_target\_to\_predicate \leftarrow target$
    $num\_targets \leftarrow num\_targets + 1$
  **end if**
  $iter$++
**end while**
```

Update of TST: When a DIP-jump commits, it updates the TST regardless of whether or not it was dynamically predicated. The TST entry for the (PC, GHR) combination is accessed and the corresponding targets are retrieved from the BTB -one per cycle- and compared to the correct target taken by the jump. If the correct target is already

[5]Figure 6 shows only the conceptual structure of the TST. In our actual implementation, the BTB index used to retrieve a target in an iteration is precomputed in parallel with the TST access. Therefore, our proposal does not increase the critical path of BTB access.

[6]Note that the values used to access the BTB to store the TST targets can conflict with real jump/branch addresses in the program, increasing aliasing and contention in the BTB. Section 7.6.2 evaluates the impact of our mechanism on performance for different BTB sizes.

stored in the BTB, the corresponding frequency counter is incremented. Otherwise, the correct target is inserted in any empty slot (i.e. for an iteration that misses in the BTB) or replacing the target with the smallest frequency counter value. Note that the TST update is not on the critical path of execution and can take multiple cycles as necessary.

The purpose of a TST entry is to provide a list of targets approximately ordered by recent execution frequency. As the saturating frequency counters are updated, if more than two counters saturate at the maximum value, it becomes impossible to distinguish the two most frequent targets. To avoid this problem, we implement a simple aging mechanism: if two of the frequency counters are found to be saturated when a TST entry is updated, all counters in the entry are right shifted by one bit. In addition to avoiding the saturation problem, this aging mechanism also demotes the targets that have not been recently used, keeping the TST content up to date for the current program phase.

5.2.3. Selecting More Than Two Targets Unlike conditional branches, indirect jumps can have more than two targets that are frequently executed. When the likelihood of having the correct target in a set of two targets is not high enough, it might be profitable to predicate multiple targets, even though the overhead of predication would be higher. If we allow predication of more than two targets, we have to select which targets and how many targets to use for each low-confidence indirect jump. The TST holds one frequency counter for each of the targets that have been more frequently used in the recent past. The aging mechanism keeps these counters representative of the current phase of the program. Therefore, it is reasonable to select the targets with higher frequency count.

To select multiple targets, the processor uses a greedy algorithm. It starts with the two targets with the highest frequency. Then, it chooses the *i*-th target in descending frequency order only if its frequency still adds significantly to the sum of the frequencies of the targets already selected. This happens when the following expression is satisfied:

$$Select\, Target_i \; if \qquad Freq_i * i >= \sum_{j=1}^{i-1} Freq_j \tag{1}$$

5.2.4. Overriding the BTB-based Target Prediction The TST has more information than a conventional BTB-based indirect jump predictor for DIP-jumps, because: (1) the TST distinguishes between targets based on the different control-flow paths leading to a jump because it is indexed with PC and branch history, while a BTB-based prediction simply provides the last seen target for the jump; (2) each entry in the TST can hold multiple targets for each combination of PC and branch history (i.e. multiple targets per jump), while a BTB-based predictor can hold only one target per jump; (3) the TST contains entries for only the DIP-jumps selected by the compiler, which reduces contention, whereas a BTB contains one entry for every indirect jump and taken conditional branch.

Our main purpose for designing the TST is to use it as a mechanism to select two or more targets for dynamic predication. However, we also found that if a TST entry contains only one target or if the most frequent target in the entry is significantly more frequent[7] than the other targets, dynamic predication provides less benefit than simply predicting the most frequent target as the target of the jump. Therefore, if one of these conditions holds when a DIP-jump is fetched, the processor, instead of entering dynamic predication mode, simply overrides the BTB-based prediction for the indirect jump and uses the single predominant target specified by the TST as the predicted target for the jump.

5.2.5. Dynamic Target Selection vs. Target Prediction Dynamic target selection using the TST is conceptually different from dynamic target prediction. A TST selects more than one target to predicate for an indirect jump. In contrast, an indirect jump predictor chooses *a single target* and uses that as the prediction for the fetched indirect jump. DIP increases the probability of having the correct target in the processor by selecting extra targets and dynamically predicating multiple paths. Nevertheless, the proposed dynamic target selection mechanism can be viewed as both a *target selector* and *target predictor* especially since we sometimes use it to override target predictions as described in the previous section. As such, we envision future indirect jump predictors designed to work directly with DIP, selecting either a single target for speculative execution, or multiple targets for dynamic predication.

5.3. Hardware Support for Predicated Execution

Once the targets to be predicated are selected, the dynamic predication process in DIP is similar to that in dynamic predication of conditional branches, which was described briefly in Section 2 and in detail by Kim et al. [31]. Here we describe the additional support required for DIP. If two targets are predicated in DIP, the additional support required is only in 1) the generation of the predicate values, 2) the handling of a possible pipeline flush when the predicate values are resolved.

When a low-confidence DIP-jump is fetched, the processor enters dpred-mode. Figure 7 shows an example of indirect jump predication with two targets. First, the processor assigns a predicate id to each path to be predicated (i.e. each selected target). Unlike in conditional branch predication in which a single predicate value (and its complement) is generated based on the branch direction, there are multiple predicate values based on the addresses of the predicated targets in DIP. The predicate value for a path is generated by comparing the predicated target address to the correct target address. The processor inserts compare micro-operations (μops) to generate predicate values for each path as shown in Figure 7b.

Unlike in conditional branch predication where one of the predicated paths is always correct, both of the predicated paths might be incorrect in DIP. As a result, the processor has to flush the whole pipeline when none of the predicated

[7]We found that a difference of at least 2 units in the 2-bit frequency counters is significant.

target addresses is the correct target. To this end, the processor generates a flush μop. The flush μop checks the predicate values and triggers a pipeline flush if none of the predicate values turns out to be TRUE (i.e., if the correct target was not predicated). If any of the predicates is TRUE, the flush μop functions as a NOP. In the example of Figure 7b, the processor inserts a flush μop to check whether or not any of the predicated targets (TARGET1 or TARGET2) is correct.

All instructions fetched during dpred-mode carry a predicate id just like in dynamic predication for a conditional branch. Since select-μops are executed only if either TARGET1 or TARGET2 is the correct target, the select-μops can be controlled by just one of the two predicates. Note that the implementation of the select-μops is the same as in dynamic predication for conditional branches. We refer the reader to [34, 31] for details on the generation and implementation of select-μops.

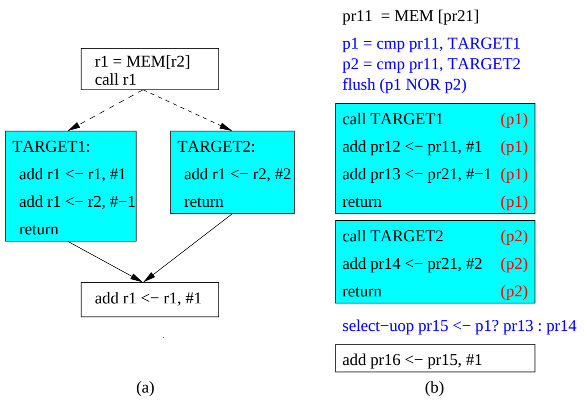

Figure 7. An example of how the instruction stream is dynamically predicated (a) control flow graph (b) dynamically predicated instructions after register renaming

5.3.1. Supporting More Than Two Targets

As we found that the predication of more than two targets does not provide significant benefits (shown and explained in Section 7.4), we only very briefly touch on hardware support for it solely for completeness. Each predicated path requires its own context: PC (program counter), GHR (global history register), RAS (return address stack), and RAT (register alias table). Since each path follows the outcomes of the branch predictor and does not fork more paths, i.e. the processor cannot be in dpred-mode for two or more nested indirect jumps at the same time, the complexity of predicating more than two targets is significantly less than the complexity of multi-path (i.e. eager) execution [37, 35]. The predication of more than two targets requires 1) storage of more frequency counters in the TST and additional combinational logic for target selection, 2) generation of more than two predicates using more than two compare instructions, 3) minor changes to the flush μop semantics to handle multiple paths, and 4) extension of the select-μop generation mechanism to handle the reconvergence of more than two paths.

5.3.2. Nested Indirect Jumps If the processor fetches another low-confidence DIP-jump during dpred-mode, it has two options: it can follow the low-confidence predicted target or it can exit dpred-mode for the earlier jump and reenter dpred-mode for the later jump. If the jumps are nested, the overhead of predicating the later DIP-jump is usually smaller than the overhead of predicating the earlier jump. Also, if the processor decides to continue in dpred-mode for the earlier jump and the later jump is mispredicted, a potentially significant part of the benefit of predication can be lost when the pipeline is flushed. Therefore, our policy (called *reentry policy*) is to exit dpred-mode for the earlier jump and *reenter dpred-mode* for the later DIP-jump. Our experimental results show that this choice provides significantly higher performance benefits (see Section 7.3).

5.3.3. Other Implementation Issues We briefly discuss other important issues in implementing DIP. Note that the same issues exist in architectures that implement static or dynamic predication for conditional branches [36, 34, 31].

Stores and Loads: A dynamically predicated store is not sent to the memory system unless its predicate is known to be TRUE. The basic rule for the forwarding logic is that a store can forward to any younger load except for stores guarded by an unresolved predicate register, which can only forward to younger loads with the same predicate id.

Interrupts and Exceptions: No special support is needed to handle interrupts and exceptions because dynamic predication state is speculative and is flushed before servicing the interrupt or exception. Predicate registers do not have to be saved and restored because they are not part of the ISA. Instructions with FALSE predicate values do not cause exceptions.

5.4. Hardware Cost and Complexity

The hardware required to dynamically predicate indirect jumps is very similar to that of the diverge-merge processor (DMP) [31, 32], which dynamically predicates conditional branches. The hardware support needed for dynamic predication (including the predicate registers, fetch/decode/rename/retirement support, and select-μops) and its cost are already described in detail by previous work [31]. We assume DIP would be cost-efficiently implemented on a baseline processor that already supports dynamic predication for conditional branches, which was shown to provide very large performance and energy benefits [31, 32]. DIP requires the following hardware modifications in addition to the required support for dynamic predication:

1. Target Selection Table (TST, Section 5.2.2): a 2K-entry, 4-way set associative table with 3 2-bit saturating counters per entry, i.e. a 1.5 KB data store and a 2.1 KB tag store (using 7-bit tags and a valid bit per entry, plus 2 bits per set for pseudo-LRU replacement).

2. A simple finite state machine to implement accessing and updating the targets in the BTB (block labeled as Control in Figure 6).

3. A 3-entry table with 32-bit constants (hash_value in Figure 6).

4. Modified predicate generation logic and flush μops (Section 5.3).

5. Optionally, support for more than 2 targets (see Section 5.3.1).

Hardware Cost: If dynamic predication hardware is already implemented for conditional branches, the cost of adding dynamic predication of indirect jumps with dynamic 2-target selection -our most efficient result- is 3.6KB of storage[8] and simple extra logic. We believe that it is not cost-effective to implement dynamic predication *only for indirect jumps*. On the contrary, dynamic predication hardware is a substrate that should be used for both conditional and indirect jumps.

5.5. ISA Support

The indirect jumps selected for dynamic predication are identified in the executable binary with a different opcode for each flavor of DIP-jump (jumps or calls). The instruction format uses one bit to indicate whether or not the jump has a return CFM point. The instruction format also includes the CFM point encoded in 16-bit 2's complement relative to the DIP-jump, which we determined is enough to encode all the CFM points we found in our set of benchmarks. When we use static target selection, the selected 32-bit targets follow the instruction in the binary. Even though these special instructions increase the code size and the pressure on the instruction cache, their impact is not significant because the number of static jumps selected for dynamic predication in our benchmarks is small (fewer than 100, as shown in Table 2).

6. Experimental Methodology

6.1. Simulation Methodology

We use an iDNA-based [3] cycle-accurate x86 simulator to evaluate dynamic indirect jump predication. Table 1 shows our baseline processor's parameters. The baseline uses a 4K-entry BTB to predict indirect jumps [40, 18]. The simulator includes a Wattch-based power model [6] using 100nm technology at 4GHz, that faithfully accounts for the power consumption of all the additional structures needed by DIP.

We evaluate DIP using benchmarks over multiple platforms. Most of the experiments are run using the 11 DaCapo benchmarks [5] (Java), Matlab R2006a (C), M5 simulator [4] (C++), and the interpreters perlbmk (C) and perlbench (C) from the SPEC CPU 2000/2006 suites. We also show results for a set of 5 SPEC CPU2000 INT benchmarks written in C, 3 SPEC CPU2006 INT benchmarks written in C, and 1 SPEC CPU2006 FP benchmark written in C++.

[8]Extra storage can be further reduced to 1.5KB with an alternative design that stores a frequency counter and a *there-is-next-target* bit directly in each BTB entry, thus eliminating the need for a separate TST. To keep the implementation conceptually simple, we do not describe this option.

Table 1. Baseline processor configuration

Front End	64KB, 2-way, 2-cycle I-cache; fetch ends at the first predicted-taken branch; fetch up to 3 conditional branches or 1 indirect branch
Branch Predictors	64KB (64-bit history, 1021-entry) perceptron branch predictor [26]; 4K-entry, 4-way BTB with pseudo-LRU replacement; 64-entry return address stack; min. branch mispred. penalty is 30 cycles
Execution Core	8-wide fetch/issue/execute/retire; 512-entry ROB; 384 physical registers; 128-entry LD-ST queue; 4-cycle pipelined wake-up and selection logic; scheduling window is partitioned into 8 sub-windows of 64 entries each
On-chip Caches	L1 D-cache: 64KB, 4-way, 2-cycle, 2 ld/st ports; L2 unified cache: 1MB, 8-way, 8 banks, 10-cycle latency; All caches: LRU repl. and 64B lines
Buses and Memory	300-cycle minimum memory latency; 32 DRAM banks; 32B-wide core-to-memory bus at 4:1 frequency ratio
Prefetcher	Stream prefetcher [42] (32 streams and 16 cacheline prefetch distance)
Dyn. pred. support	2KB (12-bit history, threshold 14) enhanced JRS confidence estimator [25], 32 predicate registers, 1 CFM register

We use those benchmarks in SPEC 2000 INT and 2006 INT/C++ suites that gain at least 5% performance with a perfect indirect jump predictor. Each benchmark is run for 200 million x86 instructions with the reference input set (SPEC CPU), small input set (DaCapo) or a custom input set (Matlab and M5)[9].

The DaCapo benchmarks are run with Sun J2SE 1.4.2_15 JRE on Windows Vista.[10] Matlab is run on Windows Vista. M5 is compiled with its default options using gcc 3.4.4, and run on Cygwin 1.5.24 on Windows Vista. All SPEC binaries are compiled with Intel's production compiler (ICC) [21] using -O3 optimizations and run on Linux Fedora Core 5. Table 2 shows the characteristics of the simulated portions of our main set of benchmarks on the baseline processor.

Table 2. Characteristics of the evaluated benchmarks

	antlr	bloat	chart	eclipse	fop	hsqldb	jython	luindex	lusearch	pmd	xalan	m5	matlab	perlbench	perlbmk	AVG
Baseline IPC	0.97	0.90	0.76	1.18	0.77	1.19	1.17	1.13	1.10	0.99	0.75	1.49	1.20	0.81	1.11	1.00
Dynamic indirect jumps (K)	4917	5390	4834	3523	7112	3054	3565	3744	4054	4557	6923	2501	2163	3614	3024	-
Indirect jump MPKI	12.50	12.40	11.60	8.50	19.70	8.30	8.60	9.10	9.80	11.40	19.20	5.60	5.70	15.40	11.30	11.27
Avg. number of dynamic targets	37.3	37.6	45.9	41.1	37.6	30.3	41.0	40.6	39.9	39.8	39.8	46.3	74.0	52.1	40.1	42.9

Table 3. DIP-related statistics for the evaluated benchmarks (CT/IT:correct/incorrect target in dpred-mode; CP/IP:correctly/incorrectly predicted)

	Metric	antlr	bloat	chart	eclipse	fop	hsqldb	jython	luindex	lusearch	pmd	xalan	m5	matlab	perlbench	perlbmk	AVG
1	ind. jumps selected for 2T DP	79	80	96	78	89	67	78	79	81	78	86	22	91	4	8	-
2	% DP instances (CT, IP)	89.0	88.7	89.6	92.1	88.4	93.1	91.6	91.7	93.5	91.3	80.1	84.8	83.0	97.6	96.1	90.0
3	% DP instances (IT, IP)	5.8	5.7	6.0	4.1	7.7	2.9	4.4	4.4	3.2	4.6	13.5	10.4	12.2	1.7	2.3	5.9
4	% DP instances (CT, CP)	4.0	4.3	3.1	2.9	2.6	3.2	3.0	2.9	2.6	3.0	3.9	3.3	2.7	0.6	1.4	2.9
5	% DP instances (IT, CP)	1.2	1.4	1.3	0.9	1.3	0.8	1.0	1.1	0.7	1.2	2.6	1.5	2.0	0.0	0.3	1.2
6	avg select-μops per DP	4.2	4.5	3.6	3.8	3.8	3.8	3.9	3.7	3.9	3.8	3.5	4.9	3.9	5.7	6.6	4.2
7	avg wrong-path instr. per DP	54.9	56.5	57.7	59.7	69.7	54.3	56.7	60.2	59.3	63.1	62.2	81.5	62.6	114.5	194.8	73.9
8	Δ pipeline flushes (%)	-47.82	-45.71	-37.74	-44.98	-46.84	-53.92	-43.86	-43.51	-49.88	-45.12	-29.06	-38.91	-22.20	-89.07	-83.73	-47.07
9	Δ fetched instr. (%)	-39.87	-41.20	-39.85	-34.28	-40.22	-41.46	-34.25	-34.40	-39.54	-37.16	-26.07	-27.20	-16.89	-62.97	-58.30	-39.32
10	Δ executed instr. (%)	-8.12	-9.46	-8.73	-5.40	-10.53	-6.31	-5.46	-5.63	-6.60	-7.18	-6.43	-2.80	-2.82	-21.64	-17.06	-8.91
11	Δ energy (%)	-26.99	-27.08	-25.32	-20.69	-28.62	-23.63	-20.15	-20.50	-24.61	-23.05	-16.02	-14.92	-5.50	-42.12	-40.18	-24.81
12	Δ energy-delay product (%)	-49.96	-49.33	-45.36	-40.17	-51.46	-44.76	-39.14	-39.56	-46.23	-43.17	-30.97	-31.33	-11.35	-66.30	-66.11	-45.54

[9]Matlab performs convolution on two images; M5 simulates the performance of gcc using its complex out-of-order processor model.

[10]At the time of this writing, iDNA [3] worked for only 7 of the 11 DaCapo benchmarks running on Sun Java SE 6 (version 1.6.0_01). The average indirect jump MPKI for these benchmarks on Java 6 is 37% *higher* than on Java 1.4, but since we cannot use the full suite, we report the results for Java 1.4. We expect DIP would perform better on Java 6.

6.2. Compilation and Profiling Methodology

Our methodology targets an adaptive JIT compiler that is able to use recent profiling information to recompile the hot methods to improve performance. For the experiments in this paper, we developed a dynamic profiling tool that collects edge profiling information and computes the CFM points during the 200M instructions preceding the simulation points for each application. The algorithms to find the CFM points are similar to those algorithms described in [33]. After profiling, our tool applies the jump selection algorithm described in Section 5.1.[11]

7. Results

7.1. Dynamic Target Distribution

Table 2 shows that the average number of dynamic targets for an indirect jump in the simulated benchmarks is 42.9. We found that in our set of object-oriented benchmarks 61% of all dynamic indirect jumps have 16 or more targets, which is significantly higher than what was reported in previous work for SPEC CPU C/C++ applications [30]. Even though indirect jumps have many targets, the most frequently executed targets (over the whole run of the program) are taken by a significant fraction of all dynamic indirect jumps, as we show in Figure 8. On average, the two most frequent targets cover 59.9% of the dynamic indirect jumps, but only 39.5% of the indirect jump mispredictions. The contribution of less frequently executed targets steadily drops. This data shows that statically selecting two targets for dynamic predication would likely not be very successful in these object-oriented applications where indirect jumps have a very large number of targets.

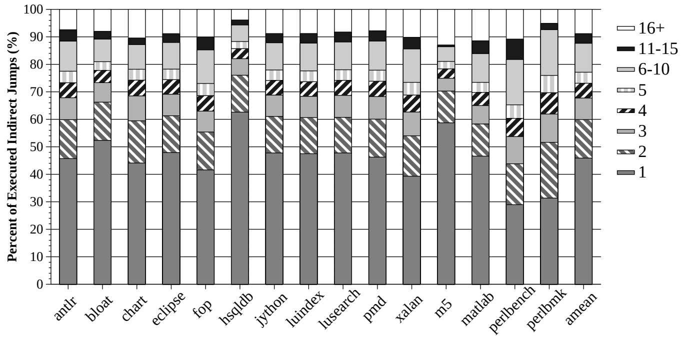

Figure 8. Fraction of dynamic indirect jumps taking the most frequently executed N targets

7.2. Performance of DIP

The first set of bars in Figure 9 shows the performance improvement of DIP over the baseline, using dynamic 2-target selection with a 3.6KB TST and all the techniques described in Section 5. The average IPC improvement of DIP is 37.8% and is analyzed in Section 7.3.

We also include five idealized experiments in Figure 9 to show the potential of DIP. The IPC improvement increases

[11]For the static target selection experiments, our dynamic profiling tool also applies the target selection algorithm described in Section 5.2.1.

to 51% if an unrealistically large TST is used (64K-entry TST with local storage for 255 targets and 32-bit frequency counters, which has a total data storage size of 128MB). If the 128MB TST always ideally provides the correct target for predication among the 2 selected targets (2T perfect target), the performance improves only by 0.2%. This means that the principles of the TST are adequate for selecting the correct target among the two that are predicated. If the DIP mechanism were used ideally only when the DIP-jump is actually mispredicted (2T perfect confidence) IPC improves by an additional 2%. The combination of perfect confidence estimation and perfect target selection (2T perfect targ./conf.) adds only an extra 0.5%, showing that the maximum potential performance benefit of 2-target DIP is 53.8%. Perfect indirect jump prediction (perfect IJP) provides 72.2% performance improvement over baseline, which is significantly higher than the maximum potential of DIP, because it does not have the overhead of dynamic predication. Our realistic implementation achieves 52% of the potential with perfect indirect jump prediction and 70% of the potential with the ideal 2-target DIP (2T perfect targ./conf.). Xalan and matlab do not get as much of the potential as the other benchmarks because the TST miss rate is significantly high (43% for both).

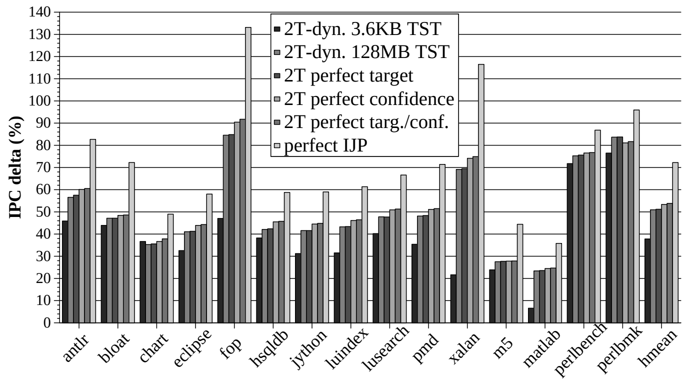

Figure 9. DIP performance and potential

7.3. Analysis of the Performance Improvement

The performance improvement of DIP comes from avoiding the full pipeline flushes caused by indirect jump mispredictions. DIP can improve performance only if it selects the correct target as one of the targets to predicate. Therefore, most of our design effort for DIP is focused on mechanisms to improve target selection. On average, DIP eliminates 47% of the pipeline flushes that occur with a BTB-based predictor (as shown in Table 3, row 8). Furthermore, the overhead of executing the extra path is low: the average number of dynamically predicated wrong-path instructions is only 73.9 (Table 3, row 7), which is significantly smaller than the instruction window size of the processor. Hence, in the steady state, dynamic predication of a mispredicted jump would result in only 73.9 instruction slots to be wasted whereas the misprediction itself would have resulted in all instruction slots in the window plus those in the front-end pipeline stages to be wasted.

The benefit of DIP depends on the combination of target selection and confidence estimation. We classify dynamic predication instances into four cases based on whether or not the correct target is predicated and whether or not the jump was actually mispredicted:

1. *Useful:* A dynamic predication instance is useful (i.e. successfully avoids a pipeline flush) if it predicates the correct target and the jump was originally mispredicted. On average, this happens for 90% of the dynamic predication instances (Table 3, row 2).

2. *Neutral:* If the jump was mispredicted but DIP does not predicate the correct target, DIP has no impact on performance. This case is no different from a misprediction because the pipeline is flushed, but it would have been flushed anyway because the jump is mispredicted. This case accounts for 5.9% of the dynamic predication instances (row 3).

3. *Moderately harmful:* If DIP decides to predicate a jump that was correctly predicted, there is performance degradation. If the correct target is one of the predicated targets, the degradation is less severe (it is only due to the overhead of executing the extra predicated path). This happens for 2.9% of the dynamic predication instances (row 4).

4. *Harmful:* The worst case is dynamically predicating a correctly-predicted jump without predicating the correct target, which introduces a new pipeline flush that would not have happened without dynamic predication. However, this worst case occurs only in 1.2% of the dynamic predication instances (row 5).

Figure 10 shows the outcomes of all executed indirect jumps with DIP: 46.8% were correctly predicted by the BTB, 39.9% were dynamically predicated and fall into one of the four cases described above, and 14.3% were mispredicted but not predicated by DIP. Hence, DIP is effective at eliminating most of the indirect jump mispredictions.

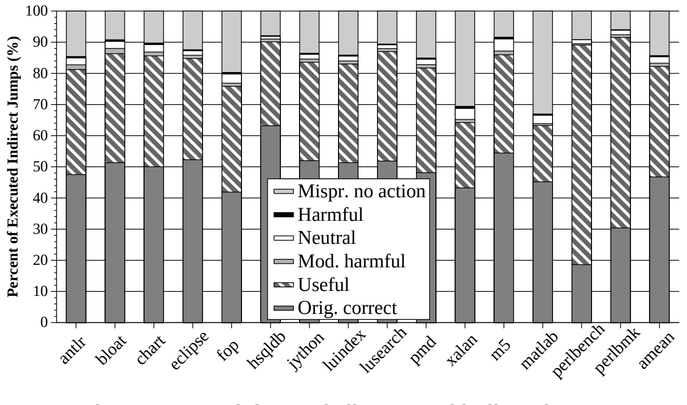

Figure 10. Breakdown of all executed indirect jumps

Effect of Different DIP Mechanisms Figure 11 shows the performance improvement due to the cumulative application of the different mechanisms included in DIP for dynamic 2-target selection. Basic DIP using only regular CFM points provides 12.5% average performance improvement. Including return CFM points slightly increases the IPC improvement to 13.9%. The reentry policy for nested indirect jumps (Section 5.3.2) significantly increases the benefit to 29% because it enables the benefit of DIP for the innermost low-confidence jumps, which are more likely to have merging control flow without being disrupted by further mispredictions than the outermost jumps. Finally, overriding the indirect jump prediction when there is one dominant target in the TST increases the average IPC improvement to 37.8% because it reduces the overhead of DIP. The last set of bars show that overriding alone, i.e. using the TST as an indirect jump predictor, provides about 70% of the benefit of full DIP.

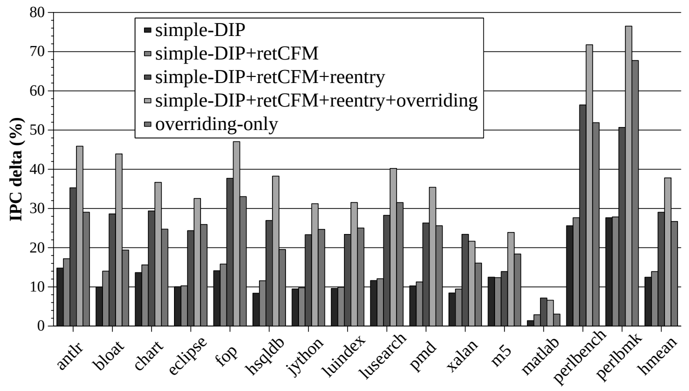

Figure 11. Performance improvement of different DIP mechanisms

7.4. Effect of Target Selection Policies

Static selection: Figure 12 shows the performance improvement of DIP over the baseline for different number of predicated targets and target selection techniques. The average IPC improvement with two statically selected targets is 6.6%. Increasing the number of static targets improves performance by up to 14.1% (for 5 targets). The 2 most frequently executed targets account for 59.9% of the executed indirect jumps (Figure 8) but only 39.5% of the indirect jump mispredictions. Even though 5 static targets cover 77% of the executions and 64% of the mispredictions, this is still not high enough to prevent most of the mispredictions. Additionally, the benefit of having the correct target is offset by the overhead of *always* predicating the extra paths. Therefore, static target selection does not provide high performance.

Dynamic selection: Dynamic 2-target selection with a 3.6KB Target Selection Table improves IPC much more significantly (by 37.8%) than static 2-target selection because the TST (1) keeps the most likely targets for the current phase and context of the program thereby increasing the probability of predicating the correct target and (2) avoids the overhead of predication when one target is dominant by overriding the jump prediction (Section 5.2.4). Increasing the maximum number of targets that can be predicated (using the dynamic target selection algorithm of Section 5.2.3) improves IPC by more than 2% only for `chart`. In the other benchmarks, there is almost no effect on IPC or there is performance degradation due to the overhead of the extra paths. The *two* most frequent targets in the recent past provide most of the benefit, as already shown by the experiment with perfect targets in Section 7.2. *We conclude that the most efficient implementation of DIP is with dynamic 2-target selection* and use this implementation in the rest of our evaluations.

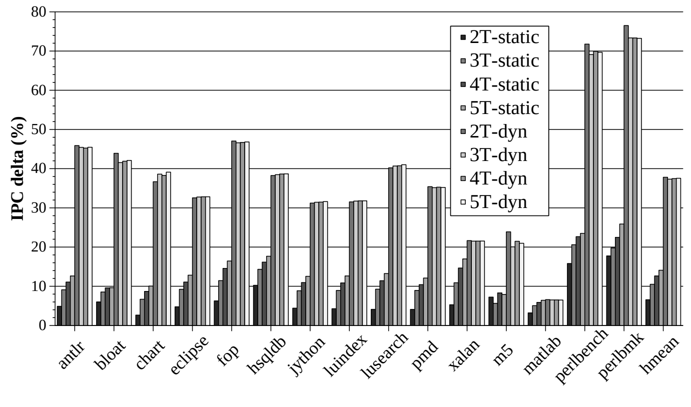

Figure 12. Performance of DIP with different target selection policies

7.5. DIP versus Indirect Jump Predictors

Figure 13(top) compares the performance of DIP with the tagged target cache (TTC) predictor [9]. Our TTC is 4-way set associative and uses full tags, but its size is computed assuming only 2-byte tags and 4-byte targets per entry, plus pseudo-LRU and valid bits. Since an entry in the TTC is created only when the BTB mispredicts, the monomorphic or easy-to-predict indirect jumps do not contend for TTC space, unlike previous work [9]. On average, DIP with a 3.6KB TST performs 6.2% better than a 12.4KB TTC and within 1.8% of a 24.8KB TTC. For four of the benchmarks, DIP performs better than a 24.8KB TTC. Figure 13(bottom) shows the IPC improvement of DIP *on a baseline with a TTC of the indicated size*. DIP improves IPC for every TTC size, from 18.6% on a processor with a 3.1KB TTC to 3.8% on a processor with a very large, 24.8KB TTC.

Figure 13(top) also shows (in the fourth bars from the left) that DIP performs 12.2% better than the recently proposed VPC predictor [30], configured to perform up to 12 prediction iterations. If VPC is used in the baseline to predict indirect jumps, DIP still improves IPC by 6.6% (Figure 13(bottom)).

Figure 13(top) also compares (in the rightmost two bars) the performance of DIP with a 3-stage cascaded predictor [13].[12] On average, DIP performs 4.5% better than an 11.3KB cascaded predictor and within 2.4% of a 22.6KB cascaded predictor. Figure 13(bottom) shows that DIP can improve performance significantly even on baseline processors with very large cascaded predictors.

Summary: Our comparisons of DIP with three of the best previously-proposed indirect jump predictors show that: 1) DIP can provide significantly higher performance than that provided by predictors with larger storage cost, 2)

[12]The size of the cascaded predictor is the sum of the sizes of the data store and tag store tables, assuming 2-byte tags and 4-byte targets, although we simulate full tags. An 11.3KB cascaded predictor performs 1.2% better than a 12.4KB TTC.

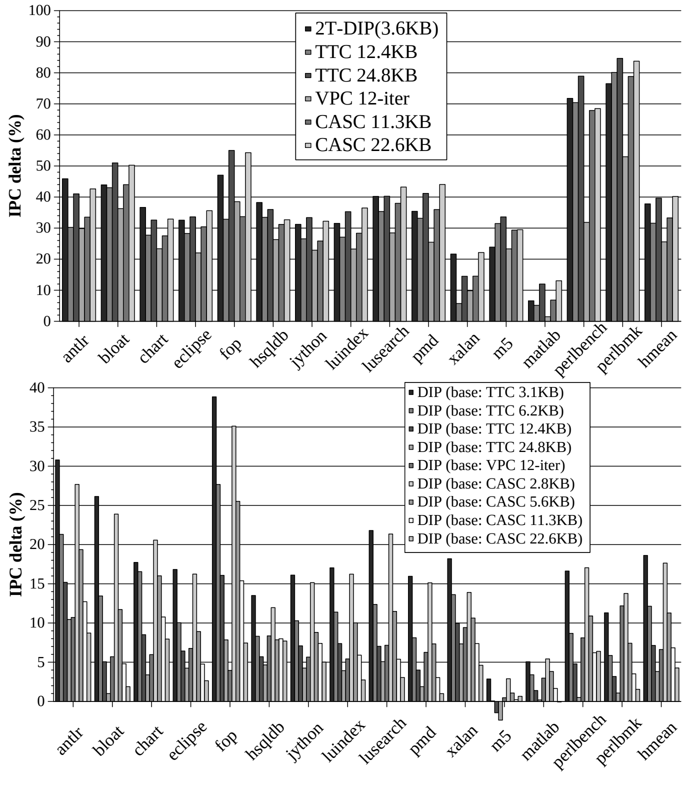

Figure 13. Performance of DIP vs. Indirect Jump Predictors

DIP can significantly improve performance even when used in conjunction with a large indirect jump predictor, and 3) DIP is very effective in reducing the performance impact of indirect jumps that are difficult to predict even with sophisticated indirect jump predictors. As such, *we conclude that DIP is an effective indirect jump handling technique that can replace or be used in conjunction with previously-proposed indirect jump predictors*.

7.6. Sensitivity to Microarchitectural Parameters

7.6.1. Less Aggressive Baseline Processor Figure 14 shows the performance of DIP along with TTC, VPC and 3-stage cascaded predictors on a less aggressive processor with 4-wide fetch/issue/retire rate, 20-stage pipeline, 128-entry instruction window, 16KB perceptron branch predictor and 200-cycle memory latency. Improving indirect jump handling on a less aggressive processor provides a smaller performance improvement due to the reduced jump misprediction penalty. However, DIP (with a 3.6KB TST) still improves performance by 25.2%, very close to the performance with a 24.8KB TTC predictor or a 22.6KB cascaded predictor.

7.6.2. BTB Sizes Table 4 shows average results for DIP with different BTB sizes from 1K to 16K entries. The performance improvement of DIP increases with BTB size because contention due to storing extra targets in the BTB for target selection becomes less of a problem. However, DIP's performance improvement is still significant (18.1%) with a small 1K-entry BTB. Our baseline 4K-entry BTB -similar to the one in Pentium 4 [18]- allows most of the

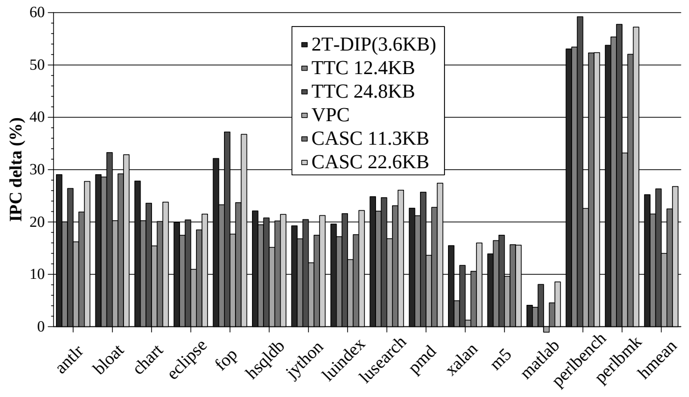

Figure 14. Performance of DIP on a less aggressive processor

benefit of DIP that can be obtained with larger BTBs.

Table 4. Effect of different BTB sizes

BTB entries (size)	Baseline			DIP 2-target		
	cond. br. BTB miss%	indi. MPKI	IPC	cond. br. BTB miss%	IPC	IPC Δ
1K (6.4 KB)	4.57%	11.68	0.95	5.89%	1.12	**18.1%**
2K (12.9 KB)	1.86%	11.40	0.98	2.53%	1.29	**30.7%**
4K (25.8 KB)	0.74%	11.27	1.00	1.14%	1.37	**37.8%**
8K (51.5 KB)	0.23%	11.20	1.00	0.45%	1.41	**41.5%**
16K (103 KB)	0.07%	11.19	1.00	0.15%	1.41	**41.2%**

7.7. Effect of Dynamic Target Selection Hardware

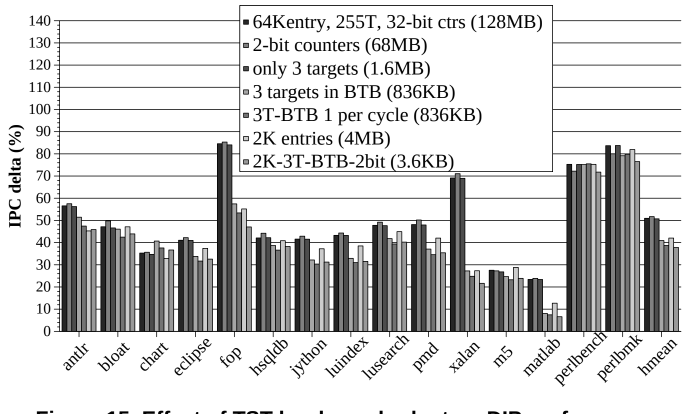

Figure 15. Effect of TST hardware budget on DIP performance

Figure 15 shows the effect of the design parameters of the Target Selection Table (TST) for dynamic 2-target selection. We start from an unrealistic TST that achieves most of the potential for perfect target selection, as discussed in Section 7.2. The rest of the experiments introduce realistic limits on the TST. Reducing the size of the counters to 2-bit saturating counters actually helps in most of the benchmarks because the aging mechanism improves the ability to track the current phase behavior. The realistic constraints that reduce the IPC improvement most significantly are: (1) storing the targets in the BTB instead of in the TST (because this creates contention for BTB entries); and (2) reducing the number of TST entries to 2K (because the TST hit rate drops from 97% to 87%). The effects of these two performance limiters mostly overlap because both the TST and the BTB use the LRU replacement policy. Since we cannot add extra ports to the BTB to access all the targets in one cycle, we model one access to the BTB per cycle, which reduces the IPC improvement by 2%. The results show that a realistic 3.6KB TST performs only 13% below the unrealistic 128MB TST. We conclude that our TST design is efficient and effective for our purposes.

Figure 16 shows the performance improvement for different TST configurations (number of entries and associativity). Our 3.6KB configuration (2K entries, 4-way set associative) is a good trade-off because it provides most of the performance of a larger TST.

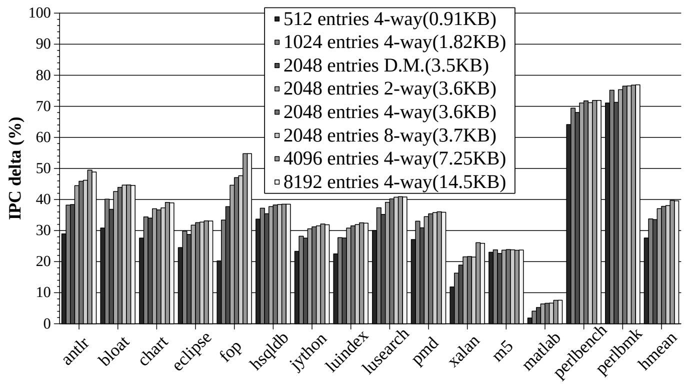

Figure 16. Performance of DIP with different TST configurations

7.8. Performance on SPEC Integer Benchmarks

Figure 17 shows the performance of DIP on the subset of SPEC CPU 2000 and 2006 benchmarks described in Section 6. Even though the SPEC benchmarks are not as indirect jump intensive as the object-oriented Java DaCapo benchmarks, DIP still increases performance by 26% on average, more than the VPC predictor and very close to a 12.4KB TTC predictor or a 22.6KB cascaded predictor.

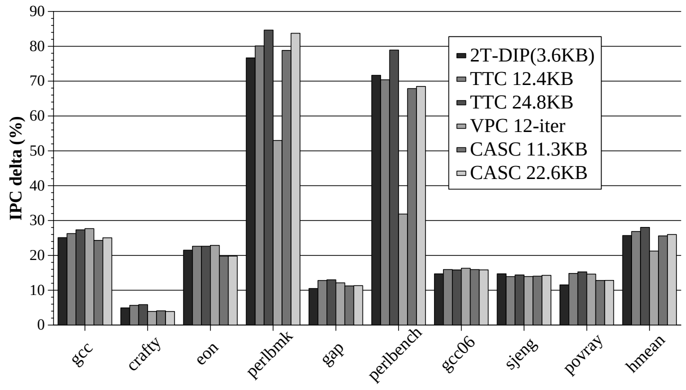

Figure 17. DIP performance on SPEC CPU integer benchmarks

7.9. Effect on Energy and Power Consumption

DIP reduces energy consumption by 24.8% (Table 3, row 11) and energy-delay product by 45.5% on average (row 12). The significant decrease in energy reduction is because of the large reduction in fetched instructions (39.3%, row 9) and executed instructions[13] (8.9%, row 10). The reduction in fetched/executed instructions is due to the elimination of pipeline flushes. When DIP eliminates a flush by predicating the correct target for an otherwise mispredicted jump, it eliminates 1) the waste of all pipeline and instruction window slots for the execution of wrong-path instructions and 2) the need to re-fetch and re-execute instructions on the control-independent path after a predicated indirect jump.

Table 5 shows a power/energy comparison of DIP and indirect jump predictors that perform close to it. DIP reduces energy consumption and energy-delay product significantly more than any of the indirect jump predictors. DIP increases maximum power slightly more than the predictors due to the hardware required for dynamic predication. However, note that this hardware can also be used to dynamically predicate conditional branches to further increase performance and reduce energy consumption. If dynamic predication is already implemented for conditional branches [31], additional structures required for DIP would increase maximum power consumption by only 1.3%. *We conclude that DIP is the most energy-efficient mechanism for handling indirect jumps.*

Table 5. Performance, power, energy comparison of DIP and indirect jump predictors

	DIP	TTC 12.4KB	VPC	Casc. 11.3KB
IPC Δ	37.8%	33.8%	26.0%	34.8%
Max power Δ	2.27%	1.06%	0.87%	1.09%
Energy Δ	-24.8%	-21.0%	-19.6%	-21.7%
Energy $\times$ Delay Δ	-45.5%	-38.9%	-40.8%	-39.9%

8. Related Work

We have already discussed related work on compiler-based predication and dynamic predication of conditional branches in Sections 2 and 3. Previously proposed static or dynamic predication approaches were not applicable to indirect jumps.

Most current processors use the BTB [40, 18] to predict the target addresses of indirect jumps. A BTB predicts the last taken target of the indirect jump as the current target and is therefore inaccurate at predicting “polymorphic” indirect jumps that frequently switch between different targets. Specialized indirect jump predictors [9, 12, 28, 39] were proposed to predict the target addresses of indirect jumps that switch between different target addresses in a predictable manner. Recently, VPC prediction [30] was proposed to use the existing conditional branch prediction hardware to predict indirect jump targets. These previous approaches work well if the target is predictable based on

[13]The number of executed instructions includes all instructions and μops introduced by the DIP mechanism: predicate definitions, flush μops and select-μops.

past history. In contrast, DIP can reduce the performance impact of an indirect jump even if the jump is difficult to predict. We have provided extensive comparisons to indirect jump predictors. Evaluations in Section 7.5 show that DIP provides larger performance and energy improvements than indirect jump predictors that use much larger hardware storage budgets.

Dependence-based pre-computation [38] improves indirect call prediction by pre-computing targets for future virtual function calls as soon as an object reference is created, avoiding a misprediction if the result of the computation is correct and ready to be used in time. In contrast, DIP does not require any pre-computation logic, and is applicable to any indirect jump.

Pure software approaches to mitigate the performance penalty of virtual function calls include the method cache in Smalltalk-80 [11], polymorphic inline caches [19] and type feedback/devirtualization [20, 24]. Devirtualization converts an indirect jump into multiple conditional branches based on extensive program analysis or accurate profiling. The benefit of devirtualization is limited by its lack of adaptivity (as shown in [30]), very much like our static target selection mechanism. Therefore, most state-of-the-art compilers either do not use devirtualization or implement a limited form of it [41]. Code replication and superinstructions [14] were proposed to improve indirect jump prediction accuracy on virtual machine interpreters. Our approach is not specific to any platform and can be used for any indirect jump.

9. Conclusion

This paper proposed the dynamic predication of indirect jumps (DIP) as a new architectural approach to improve the performance of programming language constructs implemented using indirect jumps. DIP is a cooperative hardware/software (architecture/compiler) technique that combines the strengths of both. The key idea of DIP is that the processor follows multiple target paths of a hard-to-predict indirect jump by dynamically predicating them instead of predicting only one target for the jump. This significantly improves the likelihood that the correct target path is in the processor and therefore reduces the likelihood of a full pipeline flush due to an indirect jump with multiple dynamically-exercised targets. We showed that the hardware cost of DIP is very small if dynamic predication is already implemented for conditional branches. Therefore, we believe that dynamic predication is a substrate that should be used for both conditional and indirect jumps.

We evaluated DIP on modern object-oriented applications, including the full set of Java DaCapo benchmarks. Our results show that DIP improves performance by 37.8% over a commonly-used BTB-based indirect jump predictor, while also reducing energy consumption by 24.8%. We also evaluated DIP in comparison with three previously proposed indirect jump predictors and found that DIP provides better performance and better energy efficiency, while requiring smaller hardware storage cost. As such, DIP could be an enabler that improves the performance of modular object-oriented applications that heavily make use of indirect jumps. We believe the importance of DIP will increase in the future as more programs will likely be written in object-oriented styles to reduce software development costs and to improve ease of programming.

References

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *POPL-10*, 1983.

[2] B. Alpern, A. Cocchi, S. Fink, D. Grove, and D. Lieber. Efficient implementation of Java interfaces: Invokeinterface considered harmless. In *OOPSLA*, 2001.

[3] S. Bhansali, W.-K. Chen, S. D. Jong, A. Edwards, and M. Drinic. Framework for instruction-level tracing and analysis of programs. In *VEE*, 2006.

[4] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.

[5] S. M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA'06*, 2006.

[6] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA-27*, 2000.

[7] B. Calder, D. Grunwald, and B. Zorn. Quantifying behavioral differences between C and C++ programs. *Journal of Programming Languages*, 2(4):323–351, 1995.

[8] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–523, Dec. 1985.

[9] P. Chang, E. Hao, and Y. N. Patt. Target prediction for indirect jumps. In *ISCA*, 1997.

[10] R. Cytron et al. Efficiently computing static single assignment form and the control dependence graph. *ACM TOPLAS*, 13(4):451–490, Oct. 1991.

[11] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *POPL*, 1984.

[12] K. Driesen and U. Hölzle. Accurate indirect branch prediction. In *ISCA-25*, 1998.

[13] K. Driesen and U. Hölzle. Multi-stage cascaded prediction. In *Euro-Par*, 1999.

[14] M. A. Ertl and D. Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *PLDI*, 2003.

[15] M. Farrens, T. Heil, J. E. Smith, and G. Tyson. Restricted dual path execution. Technical Report CSE-97-18, University of California at Davis, Nov. 1997.

[16] S. Gochman, R. Ronen, I. Anati, A. Berkovits, T. Kurts, A. Naveh, A. Saeed, Z. Sperber, and R. C. Valentine. The Intel Pentium M processor: Microarchitecture and performance. *Intel Technology Journal*, 7(2), May 2003.

[17] T. Heil and J. E. Smith. Selective dual path execution. Technical report, University of Wisconsin-Madison, Nov. 1996.

[18] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, Feb. 2001.

[19] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP*, 1991.

[20] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *PLDI*, 1994.

[21] Intel Corp. ICC 9.1 for Linux. `http://www.intel.com/cd/software/products/asmo-na/eng/compilers/284264.%htm`.

[22] Intel Corp. Intel Core2 Duo Desktop Processor E6600. `http://processorfinder.intel.com/details.aspx?sSpec=SL9ZL`.

[23] Intel Corp. *Intel VTune Performance Analyzers*. http://www.intel.com/vtune/.

[24] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. A study of devirtualization techniques for a java just-in-time compiler. In *OOPSLA-15*, 2000.

[25] E. Jacobsen, E. Rotenberg, and J. E. Smith. Assigning confidence to conditional branch predictions. In *MICRO-29*, 1996.

[26] D. Jiménez and C. Lin. Dynamic branch prediction with perceptrons. In *HPCA*, 2001.

[27] J. A. Joao, O. Mutlu, H. Kim, and Y. N. Patt. Dynamic predication of indirect jumps. *IEEE Computer Architecture Letters*, May 2007.

[28] J. Kalamatianos and D. R. Kaeli. Predicting indirect branches via data compression. In *MICRO-31*.

[29] R. E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, 1999.

[30] H. Kim, J. A. Joao, O. Mutlu, C. J. Lee, Y. N. Patt, and R. S. Cohn. VPC Prediction: Reducing the cost of indirect branches via hardware-based dynamic devirtualization. In *ISCA-34*, 2007.

[31] H. Kim, J. A. Joao, O. Mutlu, and Y. N. Patt. Diverge-merge processor (DMP): Dynamic predicated execution of complex control-flow graphs based on frequently executed paths. In *MICRO-39*, 2006.

[32] H. Kim, J. A. Joao, O. Mutlu, and Y. N. Patt. Diverge-merge processor: Generalized and energy-efficient dynamic predication. *IEEE Micro*, 27(1):94–104, 2007.

[33] H. Kim, J. A. Joao, O. Mutlu, and Y. N. Patt. Profile-assisted compiler support for dynamic predication in diverge-merge processors. In *CGO-5*, 2007.

[34] A. Klauser, T. Austin, D. Grunwald, and B. Calder. Dynamic hammock predication for non-predicated instruction set architectures. In *PACT*, 1998.

[35] A. Klauser, A. Paithankar, and D. Grunwald. Selective eager execution on the polypath architecture. In *ISCA-25*, 1998.

[36] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle. The Cydra 5 departmental supercomputer. *IEEE Computer*, 22:12–35, Jan. 1989.

[37] E. M. Riseman and C. C. Foster. The inhibition of potential parallelism by conditional jumps. *IEEE Transactions on Computers*, C-21(12):1405–1411, 1972.

[38] A. Roth, A. Moshovos, and G. S. Sohi. Improving virtual function call target prediction via dependence-based pre-computation. In *ICS-13*, 1999.

[39] A. Seznec and P. Michaud. A case for (partially) TAgged GEometric history length branch prediction. *JILP*, Feb. 2006.

[40] E. H. Sussenguth. *Instruction Control Sequence*. U.S. Patent 3559183, Jan. 26, 1971.

[41] D. Tarditi, July 2007. Personal communication.

[42] J. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Technical White Paper*, Oct. 2001.

[43] P. H. Wang, H. Wang, R. M. Kling, K. Ramakrishnan, and J. P. Shen. Register renaming and scheduling for dynamic execution of predicated code. In *HPCA-7*, 2001.

[44] M. Wolczko. *Benchmarking Java with the Richards benchmark*. `http://research.sun.com/people/mario/java_benchmarking/richards/richards.html`.