# VPC Prediction: Reducing the Cost of Indirect Branches via Hardware-Based Dynamic Devirtualization

*Hyesoon Kim*[¶] *José A. Joao* *Onur Mutlu*[§] *Chang Joo Lee* *Yale N. Patt* *Robert Cohn*[†]

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

**[¶]School of Computer Science**
**Georgia Institute of Technology**
**Atlanta, GA**

**[§]Computer Architecture Group**
**Microsoft Research**
**Redmond, WA**

**[†]VSSAD Group**
**Intel Corporation**
**Hudson, MA**

**TR-HPS-2007-002**
**March 2007 (Updated September 2007)**

This page is intentionally left blank.

# VPC Prediction: Reducing the Cost of Indirect Branches via Hardware-Based Dynamic Devirtualization

Hyesoon Kim¶‡ José A. Joao‡ Onur Mutlu§ Chang Joo Lee‡ Yale N. Patt‡ Robert Cohn†

¶School of Computer Science
Georgia Inst. of Technology
hyesoon@cc.gatech.edu

‡Department of ECE
University of Texas at Austin
{hyesoon, joao, cjlee, patt}@ece.utexas.edu

§Microsoft Research
onur@microsoft.com

†Intel Corporation
robert.s.cohn@intel.com

## Abstract

*Indirect branches have become increasingly common in modular programs written in modern object-oriented languages and virtual-machine based runtime systems. Unfortunately, the prediction accuracy of indirect branches has not improved as much as that of conditional branches. Furthermore, previously proposed indirect branch predictors usually require a significant amount of extra hardware storage and complexity, which makes them less attractive to implement.*

*This paper proposes a new technique for handling indirect branches, called* Virtual Program Counter (VPC) prediction*. The key idea of VPC prediction is to treat a single indirect branch as* multiple "virtual" conditional branches *in hardware for prediction purposes. Our technique predicts each of the virtual conditional branches using the existing conditional branch prediction hardware. Thus, no separate storage structure is required for predicting indirect branch targets.*

*Our comprehensive evaluation shows that VPC prediction improves average performance by 26.7% and reduces average energy consumption by 19% compared to a commonly-used branch target buffer based predictor on 12 indirect branch intensive C/C++ applications. VPC prediction achieves the performance improvement provided by at least a 12KB (and usually a 192KB) tagged target cache predictor on half of these applications. Furthermore, VPC prediction improves the average performance of the full set of object-oriented Java DaCapo applications by 21.9%, while reducing their average energy consumption by 22%. We show that VPC prediction can be used with any existing conditional branch prediction mechanism and that the accuracy of VPC prediction improves when a more accurate conditional branch predictor is used.*

## 1. Introduction

Object-oriented programs are becoming more common as more programs are written in modern high-level languages such as Java, C++, and C#. These languages support polymorphism [7], which significantly eases the development and maintenance of large modular software projects. To support polymorphism, modern languages include dynamically-dispatched function calls (i.e. virtual functions) whose targets are not known until run-time because they depend on the dynamic type of the object on which the function is called. Virtual function calls are usually implemented using indirect branch/call instructions in the instruction set architecture. Previous research has shown that modern object-oriented languages result in significantly more indirect branches than traditional C and Fortran languages [6]. Unfortunately, an indirect branch instruction is more costly on processor performance because predicting an indirect branch is more difficult than predicting a conditional branch as it requires the prediction of the target address instead of the prediction of the branch direction. Direction prediction is inherently simpler because it is a *binary* *decision* as the branch direction can take only two values (taken or not-taken), whereas indirect target prediction is an *N-ary decision* where *N* is the number of possible target addresses. Hence, with the increased use of object-oriented languages, indirect branch target mispredictions have become an important performance limiter in high-performance processors.[1] Moreover, the lack of efficient architectural support to accurately predict indirect branches has resulted in an increased performance difference between programs written in object-oriented languages and programs written in traditional languages, thereby rendering the benefits of object-oriented languages unusable by many software developers who are primarily concerned with the performance of their code [52].

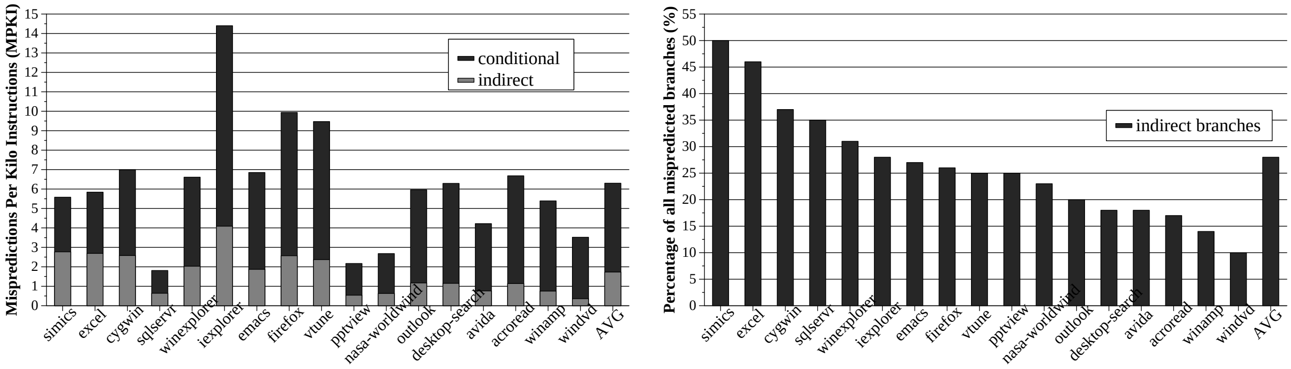

**Figure 1. Indirect branch mispredictions in Windows applications: MPKI for conditional and indirect branches (left), percentage of mispredictions due to indirect branches (right)**

Figure 1 shows the number and fraction of indirect branch mispredictions per 1K retired instructions (MPKI) in different Windows applications run on an Intel Core Duo T2500 processor [27] which includes a specialized indirect branch predictor [19]. The data is collected with hardware performance counters using VTune [28]. In the examined Windows applications, on average 28% of the branch mispredictions are due to indirect branches. In two programs, Virtutech Simics [38] and Microsoft Excel 2003, almost half of the branch mispredictions are caused by indirect branches. These results show that indirect branches cause a considerable fraction of all mispredictions even in today's relatively small-scale desktop applications.

Previously proposed indirect branch prediction techniques [9, 11, 33, 12, 13, 48] require large hardware resources to store the target addresses of indirect branches. For example, a 1024-entry gshare conditional branch predictor [40] requires only 2048 bits but a 1024-entry gshare-like indirect branch predictor (tagged target cache [9]) needs at least 2048 bytes along with additional tag storage even if the processor stores only the least significant 16 bits of an indirect branch target address in each entry.[2] As such a large hardware storage comes with an expensive increase

[1] In the rest of this paper, an "indirect branch" refers to a non-return unconditional branch instruction whose target is determined by reading a general purpose register or a memory location. We do not consider return instructions since they are usually very easy to predict using a hardware return address stack [32].

[2] With a 64-bit address space, a conventional indirect branch predictor likely requires even more hardware resources to store the target addresses [33].

in power/energy consumption and complexity, most current high-performance processors do not dedicate separate hardware but instead use the branch target buffer (BTB) to predict indirect branches [1, 22, 34], which implicitly –and usually inaccurately– assumes that the indirect branch will jump to the same target address it jumped to in its previous execution [9, 33].[3] To our knowledge, only Intel Pentium M implements specialized hardware to help the prediction of indirect branches [19], demonstrating that hardware designers are increasingly concerned with the performance impact of indirect branches. However, as we showed in Figure 1, even on a processor based on the Pentium M, indirect branch mispredictions are still relatively frequent.

In order to efficiently support polymorphism in object-oriented languages without significantly increasing complexity in the processor front-end, a simple and low-cost –yet effective– indirect branch predictor is necessary. A current high-performance processor already employs a large and accurate conditional branch predictor. Our goal is to use this existing conditional branch prediction hardware to also predict indirect branches instead of building separate, costly indirect branch prediction structures.

We propose a new indirect branch prediction algorithm: *Virtual Program Counter (VPC)* prediction. A VPC predictor treats a single indirect branch as multiple conditional branches *(virtual branches)* in hardware for prediction purposes. Conceptually, each virtual branch has its own unique target address, and the target address is stored in the BTB with a unique "fake" PC, which we call *virtual PC*. The processor uses the outcome of the existing conditional branch predictor to predict each virtual branch. The processor accesses the conditional branch predictor and the BTB with the virtual PC address of a virtual branch. If the prediction for the virtual branch is "taken," the target address provided by the BTB is predicted as the next fetch address (i.e. the predicted target of the indirect branch). If the prediction of the virtual branch is "not-taken," the processor moves on to the next virtual branch: it tries a conditional branch prediction again with a different virtual PC. The processor repeats this process until the conditional branch predictor predicts a virtual branch as taken. VPC prediction stops if none of the virtual branches is predicted as taken after a limited number of virtual branch predictions. After VPC prediction stops, the processor can stall the front-end until the target address of the indirect branch is resolved.

The VPC prediction algorithm is inspired by a compiler optimization, called *receiver class prediction optimization (RCPO)* [10, 24, 20, 5] or *devirtualization* [29]. This optimization statically converts an indirect branch to multiple direct conditional branches (in other words, it "devirtualizes" a virtual function call). Unfortunately, devirtualization requires extensive static program analysis or accurate profiling, and it is applicable to only a subset of indirect branches with a limited number of targets that can be determined statically [29]. Our proposed VPC prediction mechanism

[3] Previous research has shown that the prediction accuracy of a BTB-based indirect branch predictor, which is essentially a last-target predictor, is low (about 50%) because the target addresses of many indirect branches alternate rather than stay stable for long periods of time [9, 33].

provides the benefit of using conditional branch predictors for indirect branches without requiring static analysis or profiling by the compiler. In other words, VPC prediction *dynamically devirtualizes* an indirect branch without compiler support. Unlike compiler-based devirtualization, VPC prediction can be applied to *any indirect branch* regardless of the number and locations of its targets.

**Contributions.** The contributions of this paper are as follows:

1. To our knowledge, VPC prediction is the first mechanism that uses the existing conditional branch prediction hardware to predict the targets of indirect branches, without requiring any program transformation or compiler support.

2. VPC prediction can be applied using any current as well as future conditional branch prediction algorithm without requiring changes to the conditional branch prediction algorithm. Since VPC prediction transforms the problem of indirect branch prediction into the prediction of multiple virtual conditional branches, future improvements in conditional branch prediction accuracy can implicitly result in improving the accuracy of indirect branch prediction.

3. Unlike previously proposed indirect branch prediction schemes, VPC prediction does not require extra storage structures to maintain the targets of indirect branches. Therefore, VPC prediction provides a low-cost indirect branch prediction scheme that does not significantly complicate the front-end of the processor while providing the same performance as more complicated indirect branch predictors that require significant amounts of storage.

4. We comprehensively evaluate the performance and energy consumption of VPC prediction on both traditional C/C++ and modern object-oriented Java applications. Our results show that VPC prediction provides significant performance and energy improvements, increasing average performance by 26.7%/21.9% and decreasing energy consumption by 19%/22% respectively for 12 C/C++ and 11 Java applications. We find that the effectiveness of VPC prediction improves as the baseline BTB size and conditional branch prediction accuracy increase.

## 2. Background on Indirect Branch Prediction

We first provide a brief background on how indirect branch predictors work to motivate the similarity between indirect and conditional branch prediction. There are two types of indirect branch predictors: history-based and precomputation-based [45]. The technique we introduce in this paper utilizes history information, so we focus on history-based indirect branch predictors.

### 2.1. Why Does History-Based Indirect Branch Prediction Work?

History-based indirect branch predictors exploit information about the control-flow followed by the executing program to differentiate between the targets of an indirect branch. The insight is that the control-flow path leading to an indirect branch is strongly correlated with the target of the indirect branch [9]. This is very similar to modern conditional branch predictors, which operate on the observation that the control-flow path leading to a branch is correlated with the direction of the branch [15].

**2.1.1. A Source Code Example** The example in Figure 2 shows an indirect branch from the GAP program [16] to provide insight into why history-based prediction of indirect branch targets works. GAP implements and interprets a language that performs mathematical operations. One data structure in the GAP language is a list. When a mathematical function is applied to a list element, the program first evaluates the value of the index of the element in the list (line 13 in Figure 2). The index can be expressed in many different data types, and a different function is called to evaluate the index value based on the data type (line 10). For example, in expressions L(1), L(n), and L(n+1), the index is of three different data types: T_INT, T_VAR, and T_SUM, respectively. An indirect jump through a jump table (EvTab in lines 2, 3 and 10) determines which evaluation function is called based on the data type of the index. Consider the mathematical function L2(n) = L1(n) + L1(n+1). For each n, the program calculates three index values; Eval_VAR, Eval_SUM, and Eval_VAR functions are called respectively to evaluate index values for L1(n), L1(n+1), and L2(n). The targets of the indirect branch that determines the evaluation function of the index are therefore respectively the addresses of the two evaluation functions. Hence, the target of this indirect branch alternates between the two functions, making it unpredictable with a BTB-based last-target predictor. In contrast, a predictor that uses branch history information to predict the target easily distinguishes between the two target addresses because the branch histories followed in the functions Eval_SUM and Eval_VAR are different; hence the histories leading into the next instance of the indirect branch used to evaluate the index of the element are different. Note that a combination of the regularity in the input index expressions and the code structure allows the target address to be predictable using branch history information.

### 2.2. Previous Work on Indirect Branch Prediction

The indirect branch predictor described by Lee and Smith [36] used the branch target buffer (BTB) to predict indirect branches. This scheme predicts that the target of the current instance of the branch will be the same as the target taken in the last execution of the branch. This scheme does not work well for indirect branches that frequently switch between different target addresses. Such indirect branches are commonly used to implement virtual function calls that act on many different objects and switch statements with many ‘case’ targets that are exercised at run-time.

```
// Set up the array of function pointers (i.e. jump table)
EvTab[T_INT] = Eval_INT;  EvTab[T_VAR] = Eval_VAR;
EvTab[T_SUM] = Eval_SUM;
// ...

// EVAL evaluates an expression by calling the function
//  corresponding to the type of the expression
//  using the EvTab[] array of function pointers

#define EVAL(hd) ((*EvTab[TYPE(hd)])((hd))) /*INDIRECT*/

TypHandle  Eval_LISTELEMENT ( TypHandle hdSel ) {
    hdPos = EVAL( hdSel );
    // evaluate the index of the list element
    // check if index is valid and within bounds
    // if within bounds, access the list
    // at the given index and return the element
}
```

**Figure 2. An indirect branch example from GAP**

Therefore, the BTB-based predictor has low (about 50%) prediction accuracy [36, 9, 11, 33].

Chang et al. [9] first proposed to use branch history information to distinguish between different target addresses accessed by the same indirect branch. They proposed the “target cache,” which is similar to a two-level gshare [40] conditional branch predictor. The target cache is indexed using the XOR of the indirect branch PC and the branch history register. Each cache entry contains a target address. Each entry can be tagged, which reduces interference between different indirect branches. The tagged target cache significantly improves indirect branch prediction accuracy compared to a BTB-based predictor. However, it also requires separate structures for predicting indirect branches, increasing complexity in the processor front-end.

Later work on indirect branch prediction by Driesen and Hölzle focused on improving the prediction accuracy by enhancing the indexing functions of two-level predictors [11] and by combining multiple indirect branch predictors using a cascaded predictor [12, 13]. The cascaded predictor is a hybrid of two or more target predictors. A relatively simple first-stage predictor is used to predict easy-to-predict (single-target) indirect branches, whereas a complex second-stage predictor is used to predict hard-to-predict indirect branches. Driesen and Hölzle [13] concluded that a 3-stage cascaded predictor performed the best for a particular set of C and C++ benchmarks.

Kalamatianos and Kaeli [33] proposed predicting indirect branches via data compression. Their predictor uses prediction by partial matching (PPM) with a set of Markov predictors of decreasing size indexed by the result of hashing a decreasing number of bits from previous targets. The Markov predictor is a large set of tables where each table entry contains a single target address and bookkeeping bits. The prediction comes from the highest order table that can predict, similarly to a cascaded predictor. The PPM predictor requires significant additional hardware complexity in the indexing functions, Markov tables, and additional muxes used to select the predicted target address.

In a recent work, Seznec and Michaud [48] proposed extending their TAGE conditional branch predictor to also predict indirect branches. Their mechanism (ITTAGE) uses a tagless base predictor and a number of tagged tables (4 or 7 in the paper) indexed by an increasingly long history. The predicted target comes from the component with longer history that has a hit. This mechanism is conceptually similar to a multi-stage cascaded predictor with geometric history lengths, and therefore, it also requires significant additional storage space for indirect target addresses and significant complexity to handle indirect branches.

### 2.3. Our Motivation

All previously proposed indirect branch predictors (except the BTB-based predictor) require separate hardware structures to store target addresses in addition to the conditional branch prediction hardware. This not only requires significant die area (which translates into extra energy/power consumption), but also increases the design complexity of the processor front-end, which is already a complex and cycle-critical part of the design.[4] Moreover, many of the previously proposed indirect branch predictors are themselves complicated [12, 13, 33, 48], which further increases the overall complexity and development time of the design. For these reasons, most current processors do not implement separate structures to predict indirect branch targets.

Our goal in this paper is to design *a low-cost technique that accurately predicts indirect branch targets (by utilizing branch history information to distinguish between the different target addresses of a branch) without requiring separate complex structures for indirect branch prediction*. To this end, we propose Virtual Program Counter (VPC) prediction.

## 3. Virtual Program Counter (VPC) Prediction

### 3.1. Overview

A VPC predictor treats an indirect branch as a sequence of multiple *virtual conditional branches*.[5] Each virtual branch is predicted in sequence using the existing conditional branch prediction hardware, which consists of the direction predictor and the BTB (Figure 3). If the virtual branch is predicted to be not-taken, the VPC predictor moves on to predict the next virtual branch in the sequence. If the virtual branch is predicted to be taken, VPC prediction uses the target associated with the virtual branch in the BTB as the next fetch address, completing the prediction of the indirect branch. Note that the virtual branches are visible only to the branch prediction hardware.

### 3.2. Prediction Algorithm

The detailed VPC prediction algorithm is shown in Algorithm 1. *The key implementation issue in VPC prediction is how to distinguish between different virtual branches. Each virtual branch should access a different location in the*

[4] Using a separate predictor for indirect branch targets adds one more input to the mux that determines the predicted next fetch address. Increasing the delay of this mux can result in increased cycle time, adversely affecting the clock frequency.

[5] We call the conditional branches “virtual” because they are not encoded in the program binary. Nor are they micro-operations since they are only visible to the VPC predictor.

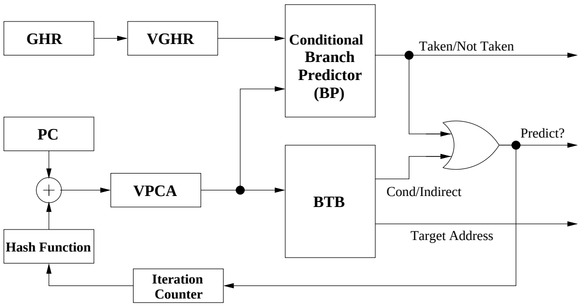

**Figure 3. High-level conceptual overview of the VPC predictor**

*direction predictor and the BTB (so that a separate direction and target prediction can be made for each branch).* To accomplish this, the VPC predictor accesses the conditional branch prediction structures with a different virtual PC address (VPCA) and a virtual global history register (GHR) value (VGHR) for each virtual branch. VPCA values are distinct for different virtual branches. VGHR values provide the context (branch history) information associated with each virtual branch.

VPC prediction is an iterative prediction process, where each iteration takes one cycle. In the first iteration (i.e. for the first virtual branch), VPCA is the same as the original PC address of the indirect branch and VGHR is the same as the GHR value when the indirect branch is fetched. If the virtual branch is predicted not-taken, the prediction algorithm moves to the next iteration (i.e. the next virtual branch) by updating the VPCA and VGHR.[6] The VPCA value for an iteration (other than the first iteration) is computed by hashing the original PC value with a randomized constant value that is specific to the iteration. In other words, $VPCA = PC \oplus HASHVAL[iter]$, where HASHVAL is a hard-coded hardware table of randomized numbers that are different from one another. The VGHR is simply left-shifted by one bit at the end of each iteration to indicate that the last virtual branch was predicted not taken.[7]

The iterative prediction process stops when a virtual branch is predicted to be taken. Otherwise, the prediction process iterates until either the number of iterations is greater than MAX_ITER or there is a BTB miss ($!pred\_target$ in Algorithm 1 means there is a BTB miss).[8] If the prediction process stops without predicting a target, the processor stalls until the indirect branch is resolved.

[6] In the first iteration, the processor does not even know that the fetched instruction is an indirect branch. This is determined only after the BTB access. If the BTB access is a hit, the BTB entry provides the type of the branch. VPC prediction algorithm continues iterating only if all of the following three conditions are satisfied: 1) the first iteration hits in the BTB, 2) the branch type indicated by the BTB entry is an indirect branch, and 3) the prediction outcome of the first iteration is not-taken.

[7] Note that VPC addresses (VPCAs) can conflict with real PC addresses in the program, thereby increasing aliasing and contention in the BTB and the direction prediction structures. The processor does not require any special action when aliasing happens. To reduce such aliasing, the processor designer should: (1) provide a good randomizing hashing function and values to generate VPCAs and (2) co-design the VPC prediction scheme and the conditional branch prediction structures carefully to minimize the effects of aliasing. Conventional techniques proposed to reduce aliasing in conditional branch predictors [40, 8] can also be used to reduce aliasing due to VPC prediction. However, our experimental results in Sections 5.6 and 7.5 show that the negative effect of VPC prediction on the BTB miss rate and conditional branch misprediction rate is tolerable.

[8] The VPC predictor can continue iterating the prediction process even if there is BTB miss. However, we found that continuing in this case does not improve the prediction accuracy. Hence, to simplify the prediction process, our VPC predictor design stops the prediction process when there is a BTB miss in any iteration.

Note that the value of MAX_ITER determines how many attempts will be made to predict an indirect branch. It also dictates how many different target addresses can be stored for an indirect branch at a given time in the BTB.

**Algorithm 1** VPC prediction algorithm

```
$iter \leftarrow 1$
$VPCA \leftarrow PC$
$VGHR \leftarrow GHR$
$done \leftarrow FALSE$
while (!$done$) do
    $pred\_target \leftarrow$ access_BTB($VPCA$)
    $pred\_dir \leftarrow$ access_conditional_BP($VPCA$, $VGHR$)
    if ($pred\_target\ and\ (pred\_dir = TAKEN)$) then
        $next\_PC \leftarrow pred\_target$
        $done \leftarrow TRUE$
    else if (!$pred\_target\ or\ (iter \geq MAX\_ITER)$) then
        $STALL \leftarrow TRUE$
        $done \leftarrow TRUE$
    end if
    $VPCA \leftarrow$ Hash($PC$, $iter$)
    $VGHR \leftarrow$ Left-Shift($VGHR$)
    $iter$++
end while
```

**3.2.1. Prediction Example** Figure 4a,b shows an example virtual function call and the corresponding simplified assembly code with an indirect branch. Figure 4c shows the virtual conditional branches corresponding to the indirect branch. Even though the static assembly code has only one indirect branch, the VPC predictor treats the indirect branch as multiple conditional branches that have different targets and VPCAs. Note that the hardware does not actually generate multiple conditional branches. The instructions in Figure 4c are shown to demonstrate VPC prediction conceptually. We assume, for this example, that MAX_ITER is 3, so there are only 3 virtual conditional branches.

```
a = s->area ();
```

(a) Source code

```
R1 = MEM[R2]
INDIRECT_CALL R1 // PC: L
```

(b) Corresponding assembly code with an indirect branch

```
iter1: cond. br TARG1 // VPCA: L
iter2: cond. br TARG2 // VPCA: VL2 = L XOR HASHVAL[1]
iter3: cond. br TARG3 // VPCA: VL3 = L XOR HASHVAL[2]
```

(c) Virtual conditional branches (for prediction purposes)

**Figure 4. VPC prediction example: source, assembly, and the corresponding virtual branches**

**Table 1. Possible VPC Predictor states and outcomes when branch in Figure 4b is predicted**

| Case | 1st iteration | | | | 2nd iteration | | | | 3rd iteration | | | | Prediction |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | inputs | | outputs | | inputs | | outputs | | input | | output | | |
| | VPCA | VGHR | BTB | BP | VPCA | VGHR | BTB | BP | VPCA | VGHR | BTB | BP | |
| 1 | L | 1111 | TARG1 | T | - | | | | - | | | | TARG1 |
| 2 | L | 1111 | TARG1 | NT | VL2 | 1110 | TARG2 | T | - | | | | TARG2 |
| 3 | L | 1111 | TARG1 | NT | VL2 | 1110 | TARG2 | NT | VL3 | 1100 | TARG3 | T | TARG3 |
| 4 | L | 1111 | TARG1 | NT | VL2 | 1110 | TARG2 | NT | VL3 | 1100 | TARG3 | NT | stall |
| 5 | L | 1111 | TARG1 | NT | VL2 | 1110 | MISS | - | - | | | | stall |

Table 1 demonstrates the five possible cases when the indirect branch in Figure 4 is predicted using VPC prediction, by showing the inputs and outputs of the VPC predictor in each iteration. We assume that the GHR is 1111 when the indirect branch is fetched. Cases 1, 2, and 3 correspond to cases where respectively the first, second, or third virtual branch is predicted taken by the conditional branch direction predictor (BP). As VPC prediction iterates, VPCA and VGHR values are updated as shown in the table. Case 4 corresponds to the case where all three of the virtual branches are predicted not-taken and therefore the outcome of the VPC predictor is a stall. Case 5 corresponds to a BTB miss for the second virtual branch and thus also results in a stall.

### 3.3. Training Algorithm

The VPC predictor is trained when an indirect branch is committed. The detailed VPC training algorithm is shown in Algorithms 2 and 3. Algorithm 2 is used when the VPC prediction was correct and Algorithm 3 is used when the VPC prediction was incorrect. The VPC predictor trains both the BTB and the conditional branch direction predictor for each predicted virtual branch. The key functions of the training algorithm are:

1. to update the direction predictor as not-taken for the virtual branches that have the wrong target (because the targets of those branches were not taken) and to update it as taken for the virtual branch, if any, that has the correct target.
2. to update the replacement policy bits of the correct target in the BTB (if the correct target exists in the BTB)
3. to insert the correct target address into the BTB (if the correct target does not exist in the BTB)

Like prediction, training is also an iterative process. To facilitate training on a correct prediction, an indirect branch carries with it through the pipeline the number of iterations performed to predict the branch ($predicted\_iter$). VPCA and VGHR values for each training iteration are recalculated exactly the same way as in the prediction algorithm. Note that only one virtual branch trains the prediction structures in a given cycle.[9]

**3.3.1. Training on a Correct Prediction** If the predicted target for an indirect branch was correct, all virtual branches except for the last one (i.e. the one that has the correct target) train the direction predictor as not-taken (as shown in Algorithm 2). The last virtual branch trains the conditional branch predictor as taken and updates the replacement policy bits in the BTB entry corresponding to the correctly-predicted target address. Note that Algorithm 2 is a special case of Algorithm 3 in that it is optimized to eliminate unnecessary BTB accesses when the target prediction is correct.

[9] It is possible to have more than one virtual branch update the prediction structures by increasing the number of write ports in the BTB and the direction predictor. We do not pursue this option as it would increase the complexity of prediction structures.

**Algorithm 2** VPC training algorithm when the branch target is correctly predicted. Inputs: $predicted\_iter$, $PC$, $GHR$

```
$iter \leftarrow 1$
$VPCA \leftarrow PC$
$VGHR \leftarrow GHR$
**while** ($iter < predicted\_iter$) **do**
    **if** ($iter == predicted\_iter$) **then**
        update_conditional_BP($VPCA$, $VGHR$, TAKEN)
        update_replacement_BTB($VPCA$)
    **else**
        update_conditional_BP($VPCA$, $VGHR$, NOT-TAKEN)
    **end if**
    $VPCA \leftarrow$ Hash(PC, $iter$)
    $VGHR \leftarrow$ Left-Shift($VGHR$)
    $iter$++
**end while**
```

**Algorithm 3** VPC training algorithm when the branch target is mispredicted. Inputs: $PC$, $GHR$, $CORRECT\_TARGET$

```
$iter \leftarrow 1$
$VPCA \leftarrow PC$
$VGHR \leftarrow GHR$
$found\_correct\_target \leftarrow FALSE$
**while** (($iter \leq MAX\_ITER$) $and$ ($found\_correct\_target = FALSE$)) **do**
    $pred\_target \leftarrow$ access_BTB($VPCA$)
    **if** ($pred\_target$ = CORRECT_TARGET) **then**
        update_conditional_BP($VPCA$, $VGHR$, TAKEN)
        update_replacement_BTB($VPCA$)
        $found\_correct\_target \leftarrow TRUE$
    **else if** ($pred\_target$) **then**
        update_conditional_BP($VPCA$, $VGHR$, NOT-TAKEN)
    **end if**
    $VPCA \leftarrow$ Hash(PC, $iter$)
    $VGHR \leftarrow$ Left-Shift($VGHR$)
    $iter$++
**end while**

/* no-target case */
**if** ($found\_correct\_target = FALSE$) **then**
    $VPCA \leftarrow$ VPCA corresponding to the virtual branch with a BTB-Miss or Least-frequently-used target among all virtual branches
    $VGHR \leftarrow$ VGHR corresponding to the virtual branch with a BTB-Miss or Least-frequently-used target among all virtual branches
    insert_BTB($VPCA$, CORRECT_TARGET)
    update_conditional_BP($VPCA$, $VGHR$, TAKEN)
**end if**
```

**3.3.2. Training on a Wrong Prediction** If the predicted target for an indirect branch was wrong, there are two misprediction cases: (1) *wrong-target*: one of the virtual branches has the correct target stored in the BTB but the direction predictor predicted that branch as not-taken, (2) *no-target*: none of the virtual branches has the correct target stored in the BTB so the VPC predictor could not have predicted the correct target. In the *no-target* case, the correct target address needs to be inserted into the BTB.

To distinguish between *wrong-target* and *no-target* cases, the training logic accesses the BTB for each virtual branch (as shown in Algorithm 3).[10] If the target address stored in the BTB for a virtual branch is the same as the correct target address of the indirect branch (*wrong-target* case), the direction predictor is trained as taken and the replacement policy bits in the BTB entry corresponding to the target address are updated. Otherwise, the direction predictor is trained as not-taken. Similarly to the VPC prediction algorithm, when the training logic finds a virtual branch with the correct target address, it stops training.

If none of the iterations (i.e. virtual branches) has the correct target address stored in the BTB, the training logic inserts the correct target address into the BTB. One design question is what VPCA/VGHR values should be used for the newly inserted target address. Conceptually, the choice of VPCA value determines the *order* of the newly inserted virtual branch among all virtual branches. To insert the new target in the BTB, our current implementation of the training algorithm uses the VPCA/VGHR values corresponding to the virtual branch that missed in the BTB. If none of the virtual branches missed in the BTB, our implementation uses the VPCA/VGHR values corresponding to the virtual branch whose BTB entry has the smallest least frequently used (LFU) value. Note that the virtual branch that missed in the BTB or that has the smallest LFU-value in its BTB entry can be determined easily while the training algorithm iterates over virtual branches (However, we do not show this computation in Algorithm 3 to keep the algorithm more readable).[11]

### 3.4. Supporting Multiple Iterations per Cycle

The iterative prediction process can take multiple cycles. The number of cycles needed to make an indirect branch prediction with a VPC predictor can be reduced if the processor already supports the prediction of multiple conditional branches in parallel [53]. The prediction logic can perform the calculation of VPCA values for multiple iterations in parallel since VPCA values do not depend on previous iterations. VGHR values for multiple iterations can also be calculated in parallel assuming that previous iterations were “not taken” since the prediction process stops when an iteration results in a “taken” prediction. Section 5.4 evaluates the performance impact of performing multiple prediction iterations in parallel.

[10] Note that these extra BTB accesses for training are required only on a misprediction and they do not require an extra BTB read port. An extra BTB access holds only one BTB bank per training-iteration. Even if the access results in a bank conflict with the accesses from the fetch engine for all the mispredicted indirect branches, we found that the performance impact is negligible due to the low frequency of indirect branch mispredictions in the VPC prediction mechanism.

[11] This scheme does not necessarily find and replace the least frequently used of the targets corresponding to an indirect branch – this is difficult to implement as it requires keeping LFU information on a per-indirect branch basis across different BTB sets. Rather, our scheme is an approximation that replaces the target that has the lowest value for LFU-bits (corresponding to the LFU within a set) stored in the BTB entry, assuming the baseline BTB implements an LFU-based replacement policy. Other heuristics are possible to determine the VPCA/VGHR of a new target address (i.e. new virtual branch). We experimented with schemes that select among the VPCA/VGHR values corresponding to the iterated virtual branches randomly, or based on the recency information that could be stored in the corresponding BTB entries and found that LFU performs best with LRU/random selection a close second/third (see Section 5.5 for a quantitative evaluation).

### 3.5. Pipelining the VPC Predictor

So far our discussion assumed that conditional branch prediction structures (the BTB and the direction predictor) can be accessed in a single processor clock cycle. However, in some modern processors, access of the conditional branch prediction structures takes multiple cycles. To accommodate this, the VPC prediction process needs to be pipelined. We briefly show that our mechanism can be trivially adjusted to accommodate pipelining.

In a pipelined implementation of VPC prediction, the next iteration of VPC prediction is started in the next cycle without knowing the outcome of the previous iteration in a pipelined fashion. In other words, consecutive VPC prediction iterations are fed into the pipeline of the conditional branch predictor one after another, one iteration per cycle. Pipelining VPC prediction is similar to supporting multiple iterations in parallel. As explained in Section 3.4, the VPCA value of a later iteration is not dependent on previous iterations; hence, VPCA values of different iterations are computed independently. The VGHR value of a later iteration is calculated assuming that previous iterations were "not taken" since the VPC prediction process stops anyway when an iteration results in a "taken" prediction. If it turns out that an iteration is not needed because a previous iteration was predicted as "taken," then the later iterations in the branch predictor pipeline are simply discarded when they produce a prediction. As such, VPC prediction naturally yields itself to pipelining without significant hardware modifications.

### 3.6. Hardware Cost and Complexity

The extra hardware required by the VPC predictor on top of the existing conditional branch prediction scheme is as follows:

1. Three registers to store $iter$, $VPCA$, and $VGHR$ for prediction purposes (Algorithm 1).
2. A hard-coded table, $HASHVAL$, of 32-bit randomized values. The table has $MAX\_ITER$ number of entries. Our experimental results show that $MAX\_ITER$ does not need to be greater than 16. The table is dual-ported to support one prediction and one update concurrently.
3. A $predicted\_iter$ value that is carried with each indirect branch throughout the pipeline. This value cannot be greater than $MAX\_ITER$.
4. Three registers to store $iter$, $VPCA$, and $VGHR$ for training purposes (Algorithms 2 and 3).
5. Two registers to store the $VPCA$ and $VGHR$ values that may be needed to insert a new target into the BTB (for the *no-target* case in Algorithm 3).

Note that the cost of the required storage is very small. Unlike previously proposed history-based indirect branch predictors, no large or complex tables are needed to store the target addresses. Instead, target addresses are naturally stored in the existing BTB.

The combinational logic needed to perform the computations required for prediction and training is also simple. Actual PC and GHR values are used to access the branch prediction structure in the first iteration of indirect branch prediction. While an iteration is performed, the VPCA and VGHR values for the next iteration are calculated and loaded into the corresponding registers. Therefore, updating VPCA and VGHR for the next iterations is not on the critical path of the branch predictor access.

The training of the VPC predictor on a misprediction may slightly increase the complexity of the BTB update logic because it requires multiple iterations to access the BTB. However, the VPC training logic needs to access the BTB multiple times only on a target misprediction, which is relatively infrequent, and the update logic of the BTB is not on the critical path of instruction execution. If needed, pending BTB and branch predictor updates due to VPC prediction can be buffered in a queue to be performed in consecutive cycles (Note that such a queue to update conditional branch prediction structures already exists in some modern processor implementations with limited number of read/write ports in the BTB or the direction predictor [39]).

## 4. Experimental Methodology

We use a Pin-based [37] cycle-accurate x86 simulator to evaluate VPC prediction. The parameters of our baseline processor are shown in Table 2. The baseline processor uses the BTB to predict indirect branches [36].

**Table 2. Baseline processor configuration**

| | |
|---|---|
| Front End | 64KB, 2-way, 2-cycle I-cache; fetch ends at the first predicted-taken br.; fetch up to 3 conditional branches or 1 indirect branch |
| Branch Predictors | 64KB (64-bit history, 1021-entry) perceptron branch predictor [30];<br>4K-entry, 4-way BTB with pseudo-LFU replacement;<br>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;<br>128-entry LD-ST queue; 4-cycle pipelined wake-up and selection logic;<br>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;<br>L2 unified cache: 1MB, 8-way, 8 banks, 10-cycle latency;<br>All caches use LRU replacement and have 64B line size |
| Buses and Memory | 300-cycle minimum memory latency; 32 memory banks;<br>32B-wide core-to-memory bus at 4:1 frequency ratio |
| Prefetcher | Stream prefetcher with 32 streams and 16 cache line prefetch distance (lookahead) [51] |

The experiments are run using 5 SPEC CPU2000 INT benchmarks, 5 SPEC CPU2006 INT/C++ benchmarks, and 2 other C++ benchmarks. We chose those benchmarks in SPEC INT 2000 and 2006 INT/C++ suites that gain at least 5% performance with a perfect indirect branch predictor. Table 3 provides a brief description of the other two C++ benchmarks.

**Table 3. Evaluated C++ benchmarks that are not included in SPEC CPU 2000 or 2006**

| | |
|---|---|
| ixx | translator from IDL (Interface Definition Language) to C++ |
| richards | simulates the task dispatcher in the kernel of an operating system [52] |

We use Pinpoints [44] to select a representative simulation region for each benchmark using the reference input set. Each benchmark is run for 200 million x86 instructions. Table 4 shows the characteristics of the examined benchmarks on the baseline processor. All binaries are compiled with Intel's production compiler (ICC) [26] with the -O3 optimization level.

**Table 4. Characteristics of the evaluated benchmarks:** language and type of the benchmark (Lang/Type), baseline IPC (BASE IPC), potential IPC improvement with perfect indirect branch prediction (PIBP IPC $\Delta$), static number of indirect branches (Static IB), dynamic number of indirect branches (Dyn. IB), indirect branch prediction accuracy (IBP Acc), indirect branch mispredictions per kilo instructions (IB MPKI), conditional branch mispredictions per kilo instructions (CB MPKI). gcc06 is 403.gcc in CPU2006 and gcc is 176.gcc in CPU2000.

| | gcc | crafty | eon | perlbmk | gap | perlbench | gcc06 | sjeng | namd | povray | richards | ixx | AVG |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Lang/Type | C/int | C/int | C++/int | C/int | C/int | C/int | C/int | C/int | C++/fp | C++/fp | C++/int | C++/int | - |
| BASE IPC | 1.20 | 1.71 | 2.15 | 1.29 | 1.29 | 1.18 | 0.66 | 1.21 | 2.62 | 1.79 | 0.91 | 1.62 | 1.29 |
| PIBP IPC $\Delta$ | 23.0% | 4.8% | 16.2% | 105.5% | 55.6% | 51.7% | 17.3% | 18.5% | 5.4% | 12.1% | 107.1% | 12.8% | 32.5% |
| Static IB | 987 | 356 | 1857 | 864 | 1640 | 1283 | 1557 | 369 | 678 | 1035 | 266 | 1281 | - |
| Dyn. IB | 1203K | 195K | 1401K | 2908K | 3454K | 1983K | 1589K | 893K | 517K | 1148K | 4533K | 252K | - |
| IBP Acc (%) | 34.9 | 34.1 | 72.2 | 30.0 | 55.3 | 32.6 | 43.9 | 28.8 | 83.3 | 70.8 | 40.9 | 80.7 | 50.6 |
| IB MPKI | 3.9 | 0.6 | 1.9 | 10.2 | 7.7 | 6.7 | 4.5 | 3.2 | 0.4 | 1.7 | 13.4 | 1.4 | 4.63 |
| CB MPKI | 3.0 | 6.1 | 0.2 | 0.9 | 0.8 | 3.0 | 3.7 | 9.5 | 1.1 | 2.1 | 1.4 | 4.2 | 3.0 |

For completeness, Table 5 shows the sensitivity of the remaining SPEC CPU2000 and CPU2006 integer benchmarks to perfect indirect branch prediction. Since indirect branches do not significantly affect the performance of these applications, VPC prediction neither improves nor degrades their performance.

**Table 5. Characteristics of the remaining SPEC CPU2000 INT and SPEC CPU2006 INT/C++ benchmarks:** baseline IPC, IPC with perfect indirect branch prediction (PIBP IPC), IPC when VPC prediction is used (VPC IPC). lib., xal., omn. are abbreviations for libquantum, xalancbmk, and omnetpp respectively.

| | SPEC CPU2000 INT | | | | | | | SPEC CPU2006 INT | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | gzip | vpr | mcf | parser | vortex | bzip2 | twolf | bzip2 | mcf | gobmk | hmmer | lib. | h264ref | omn. | astar | xal. | dealII | soplex |
| BASE IPC | 0.87 | 1.00 | 0.17 | 1.26 | 1.14 | 1.10 | 0.90 | 1.32 | 0.17 | 0.98 | 1.30 | 3.83 | 1.78 | 0.50 | 0.52 | 0.76 | 2.74 | 1.46 |
| PIBP IPC | 0.87 | 1.00 | 0.17 | 1.26 | 1.15 | 1.10 | 0.90 | 1.32 | 0.17 | 0.98 | 1.30 | 3.83 | 1.79 | 0.51 | 0.52 | 0.80 | 2.76 | 1.46 |
| VPC IPC | 0.87 | 1.00 | 0.17 | 1.26 | 1.14 | 1.10 | 0.90 | 1.32 | 0.17 | 0.98 | 1.30 | 3.83 | 1.79 | 0.50 | 0.52 | 0.78 | 2.75 | 1.46 |

# 5. Results

## 5.1. Dynamic Target Distribution

Figure 5 shows the distribution of the number of dynamic targets for executed indirect branches. In eon, gap, and ixx, more than 40% of the executed indirect branches have only one target. These single-target indirect branches are easily predictable with a simple BTB-based indirect branch predictor. However, in gcc (50%), crafty (100%), perlbmk (94%), perlbench (98%), sjeng (100%) and povray (97%), over 50% of the dynamic indirect branches have more than 5 targets. On average, 51% of the dynamic indirect branches in the evaluated benchmarks have more than 5 targets.

## 5.2. Performance of VPC Prediction

Figure 6 (left) shows the performance improvement of VPC prediction over the baseline BTB-based predictor when MAX_ITER is varied from 2 to 16. Figure 6 (right) shows the indirect branch MPKI in the baseline and with VPC prediction. In eon, gap, and namd, where over 60% of all executed indirect branches have at most 2 unique targets (as shown in Figure 5), VPC prediction with MAX_ITER=2 eliminates almost all indirect branch mispredictions. Almost

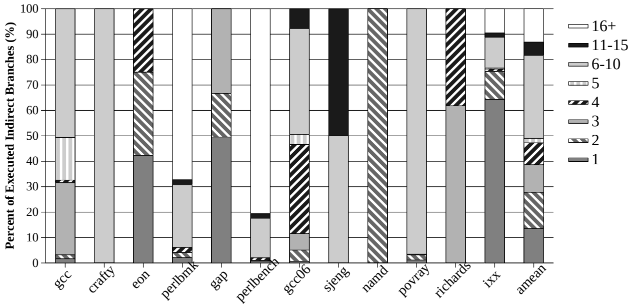

**Figure 5. Distribution of the number of dynamic targets across executed indirect branches**

all indirect branches in richards have 3 or 4 different targets. Therefore, when the VPC predictor can hold 4 different targets per indirect branch (MAX_ITER=4), indirect branch MPKI is reduced to only 0.7 (from 13.4 in baseline and 1.8 with MAX_ITER=2). The performance of only perlbmk and perlbench continues to improve significantly as MAX_ITER is increased beyond 6, because at least 65% of the indirect branches in these two benchmarks have at least 16 dynamic targets (This is due to the large switch-case statements in perl that are used to parse and pattern-match the input expressions. The most frequently executed/mispredicted indirect branch in perlbench belongs to a switch statement with 57 static targets). Note that even though the number of mispredictions can be further reduced when MAX_ITER is increased beyond 12, the performance improvement actually decreases for perlbench. This is due to two reasons: (1) storing more targets in the BTB via a larger MAX_ITER value starts creating conflict misses, (2) some correct predictions take longer when MAX_ITER is increased, which increases the idle cycles in which no instructions are fetched.

On average, VPC prediction improves performance by 26.7% over the BTB-based predictor (when MAX_ITER=12), by reducing the average indirect branch MPKI from 4.63 to 0.52. Since a MAX_ITER value of 12 provides the best performance, most later experiments in this section use MAX_ITER=12. We found that using VPC prediction does not significantly impact the prediction accuracy of conditional branches in the benchmark set we examined as shown in Table 7.

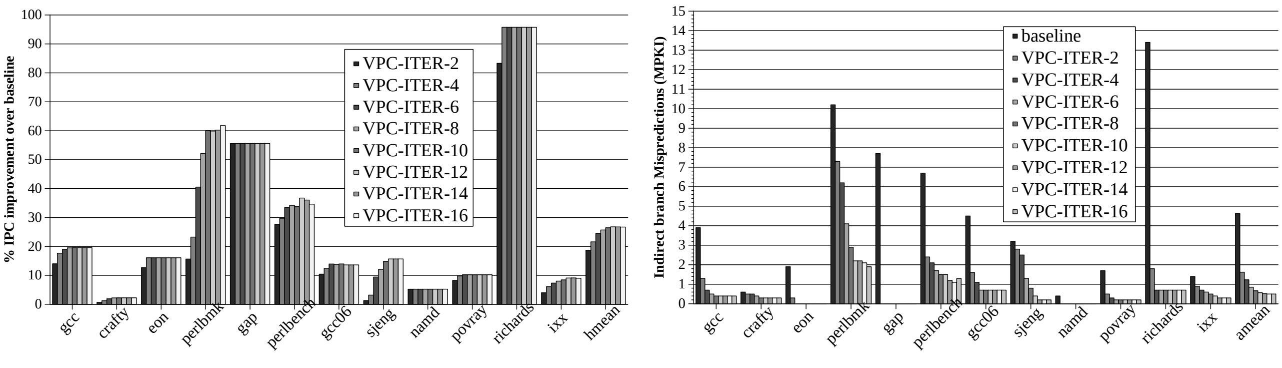

**Figure 6. Performance of VPC prediction: IPC improvement (left), indirect branch MPKI (right)**

Figure 7 shows the distribution of the number of iterations needed to generate a correct target prediction. On average 44.6% of the correct predictions occur in the first iteration (i.e. zero idle cycles) and 81% of the correct predictions occur within three iterations. Only in perlbmk and sjeng more than 30% of all correct predictions require at least 5 iterations. Hence, most correct predictions are performed quickly resulting in few idle cycles during which the fetch engine stalls.

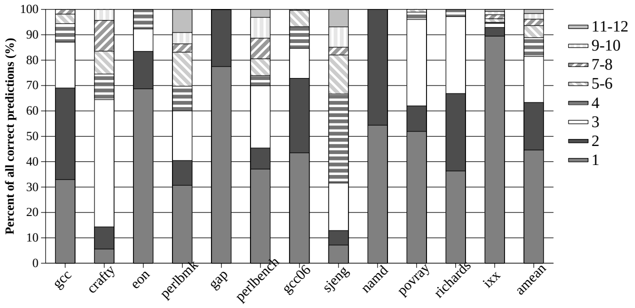

Figure 7. Distribution of the number of iterations (for correct predictions) (MAX_ITER=12)

### 5.3. Comparisons with Other Indirect Branch Predictors

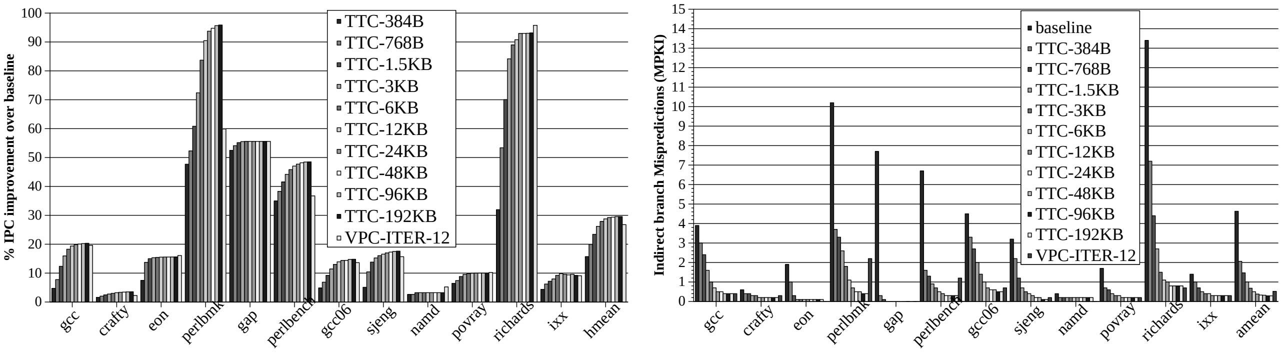

Figure 8. Performance of VPC prediction vs. tagged target cache: IPC (left), MPKI (right)

Figure 8 compares the performance and MPKI of VPC prediction with the tagged target cache (TTC) predictor [9]. The size of the 4-way TTC predictor is calculated assuming 4-byte targets and 2-byte tags for each entry.[12] On average, VPC prediction provides the performance provided by a 3-6KB TTC predictor. However, as shown in Table 6, in six benchmarks, the VPC predictor performs at least as well as a 12KB TTC (and on 4 benchmarks better than a 192KB TTC). As shown in Table 6, the size of TTC that provides equivalent performance is negatively correlated with the average number of dynamic targets for each indirect branch in a benchmark: the higher the average number of targets the smaller the TTC that performs as well as VPC (e.g. in crafty, perlbmk, and perlbench). This is because TTC provides separate storage to cache the large number of dynamic targets in addition to the BTB whereas VPC

[12] Note that we simulated full 8-byte tags for TTC and hence our performance results reflect full tags, but we assume that a realistic TTC will not be implemented with full tags so we do not penalize it in terms of area cost. A target cache entry is allocated only on a BTB misprediction for an indirect branch. Our results do not take into account the increase in cycle time that might be introduced due to the addition of the TTC predictor into the processor front-end.

prediction uses only the available BTB space. As the average number of targets increases, the contention for space in the BTB also increases, and reducing this contention even with a relatively small separate structure (as TTC does) provides significant performance gains.

**Table 6. The sizes of tagged target cache (TTC) and cascaded predictors that provide the same performance as the VPC predictor (MAX_ITER=12) in terms of IPC**

| | gcc | crafty | eon | perlbmk | gap | perlbench | gcc06 | sjeng | namd | povray | richards | ixx |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| TTC size (B) | 12KB | 1.5KB | >192KB | 1.5KB | 6KB | 512B | 12KB | 3KB | >192KB | >192KB | >192KB | 3KB |
| cascaded size (B) | >176KB | 2.8KB | >176KB | 2.8KB | 11KB | 1.4KB | 44KB | 5.5KB | >176KB | >176KB | >176KB | >176KB |
| avg. # of targets | 6.1 | 8.0 | 2.1 | 15.6 | 1.8 | 17.9 | 5.8 | 9.0 | 2.0 | 5.9 | 3.4 | 4.1 |

Figure 9 compares the performance of VPC prediction with a 3-stage cascaded predictor [12, 13]. On average, VPC prediction provides the same performance improvement as a 22KB cascaded predictor. As shown in Table 6, in six benchmarks, VPC prediction provides the performance of at least a 176KB cascaded predictor.[13]

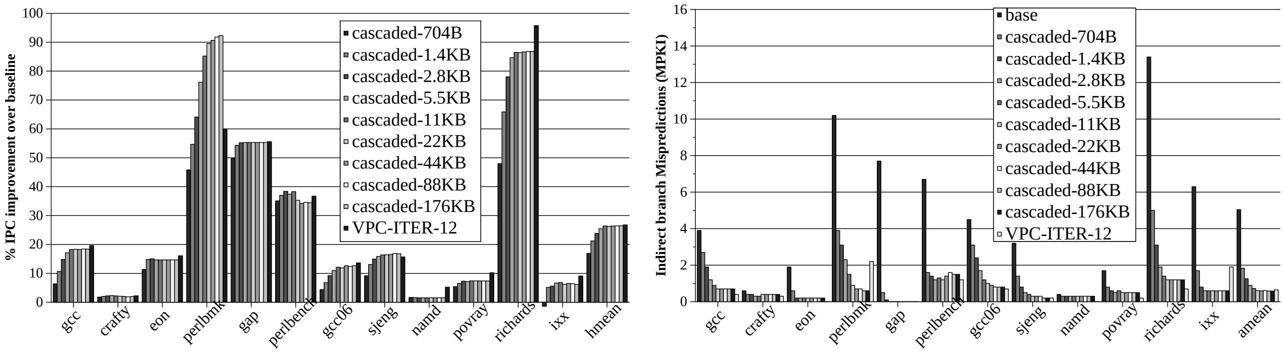

**Figure 9. Performance of VPC prediction vs. cascaded predictor: IPC (left), MPKI (right)**

## 5.4. Effect of VPC Prediction Delay

So far we have assumed that a VPC predictor can predict a single virtual branch per cycle. Providing the ability to predict multiple virtual branches per cycle (assuming the underlying conditional branch predictor supports this) would reduce the number of idle cycles spent during multiple VPC prediction iterations. Figure 10 shows the performance impact when multiple iterations can take only one cycle. Supporting, unrealistically, even 10 prediction iterations per cycle further improves the performance benefit of VPC prediction by only 2.2%. As we have already shown in Figure 7, only 19% of all correct predictions require more than 3 iterations. Therefore, supporting multiple iterations per cycle does not provide significant improvement. We conclude that, to simplify the design, the VPC predictor can be implemented to support only one iteration per cycle.

[13] We found that a 3-stage cascaded predictor performs slightly worse than an equally-sized TTC predictor. This is because the number of static indirect branches in the evaluated benchmarks is relatively small (10-20) and a cascaded predictor performs better than a TTC when there is a larger number of static branches [12, 13].

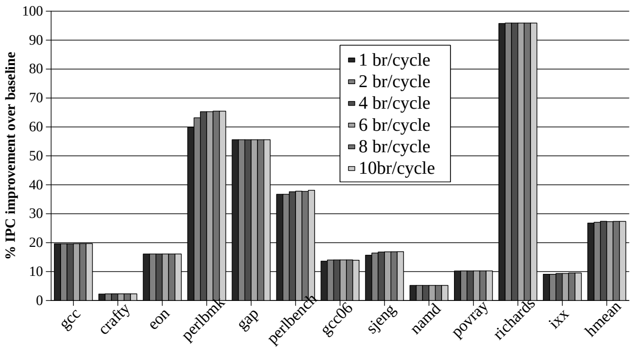

**Figure 10. Performance impact of supporting multiple VPC prediction iterations per cycle**

## 5.5. Effect of VPC Training: Where to Insert the Correct Target Address

In Section 3.3.2, we described how the VPC training algorithm inserts the correct target into the BTB if the VPC prediction was wrong. Where the correct target is inserted in the BTB with respect to other targets of the branch could affect performance because 1) it determines which target will be replaced by the new target and 2) it affects the “order” of appearance of targets in a future VPC prediction loop. This section evaluates different policies for inserting a target address into the BTB upon a VPC misprediction.

Figure 11 shows the performance improvement provided by four different policies we examine. *Naive-Insert-MAXITER* inserts the target address into the BTB without first checking whether or not it already exists in the BTB entries corresponding to the virtual branches. The target address is inserted into the first available virtual branch position, i.e. that corresponding to a virtual branch that missed in the BTB. If none of the virtual branches had missed in the BTB, the target is always inserted in the MAX_ITER position. The benefit of this mechanism is that it does not require the VPC training logic to check all the BTB entries corresponding to the virtual branches; hence it is simpler to implement. The disadvantage is it increases the redundancy of target addresses in the BTB (hence the area-efficiency of the BTB) since the target address of each virtual branch is not necessarily unique.

The other three policies we examine require each virtual branch to have a unique target address, but differ in *which* virtual branch they replace if the VPC prediction was wrong and neither the correct target of the indirect branch nor an empty virtual branch slot corresponding to the indirect branch was found in the BTB. *Unique-Random* replaces a BTB entry randomly among all the virtual branches. *Unique-LRU* replaces the target address corresponding to the virtual branch whose entry has the smallest least-recently-used (LRU) value. *Unique-LFU* is the default scheme we described in Section 3.3.2, which replaces the target address corresponding to the virtual branch whose entry has the smallest LFU-value.

According to Figure 11, the performance of most benchmarks –except perlbmk, perlbench, and sjeng– is not sensitive to the different training policies. Since the number of dynamic targets per branch is very high in perlbmk, perlbench, and sjeng (shown in Figure 5 and Table 6), the contention for virtual branch slots in the BTB is high. For our set of benchmarks, the *Unique-LFU* scheme provides the highest performance (1% and 2% better than respectively *Unique-LRU* and *Unique-Random*). We found that frequently used targets in the recent past are more likely to be used in the near future and therefore it is better to replace less frequently used target addresses. Therefore, we have chosen the *Unique-LFU* scheme as our default VPC training scheme.

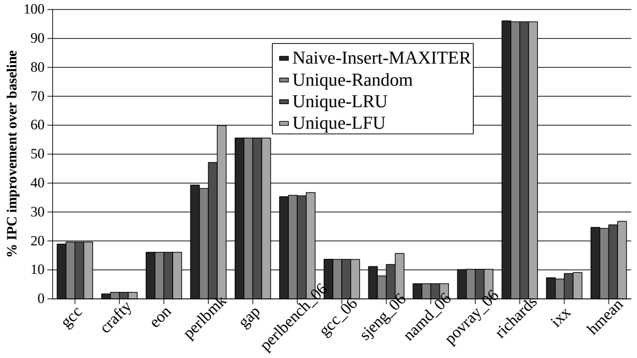

**Figure 11. Performance impact of different VPC training schemes**

### 5.6. Sensitivity of VPC Prediction to Microarchitecture Parameters

**5.6.1. Different Conditional Branch Predictors** We evaluated VPC prediction with various baseline conditional branch predictors. Figure 12 compares the performance of the TTC predictor and the VPC predictor on a baseline processor with a 64KB O-GEHL predictor [47]. On average, VPC prediction improves performance by 31% over the baseline with an O-GEHL predictor and outperforms a 12KB TTC predictor. Figure 13 shows that VPC prediction improves performance by 23.8% on a baseline processor with a 64KB gshare [40] predictor. These results show that VPC prediction can be used with other conditional branch prediction mechanisms without changing the VPC prediction algorithm solely because indirect branches are treated the same way as “multiple” conditional branches.

Table 7 summarizes the results of our comparisons. Reducing the conditional branch misprediction rate via a better predictor results in also reducing the indirect branch misprediction rate with VPC prediction. Hence, as the baseline conditional branch predictor becomes better, the performance improvement provided by VPC prediction increases. We conclude that, with VPC prediction, any research effort that improves conditional branch prediction accuracy will likely result in also improving indirect branch prediction accuracy – without requiring the significant extra effort to design complex and specialized indirect branch predictors.

**Table 7. Effect of different conditional branch predictors**

| Cond. BP | Baseline | | | VPC prediction | | |
|---|---|---|---|---|---|---|
| | cond. MPKI | indi. MPKI | IPC | cond. MPKI | indi. MPKI | IPC Δ |
| gshare | **3.70** | 4.63 | 1.25 | 3.78 | **0.65** | **23.8%** |
| perceptron | **3.00** | 4.63 | 1.29 | 3.00 | **0.52** | **26.7%** |
| O-GEHL | **1.82** | 4.63 | 1.37 | 1.84 | **0.32** | **31.0%** |

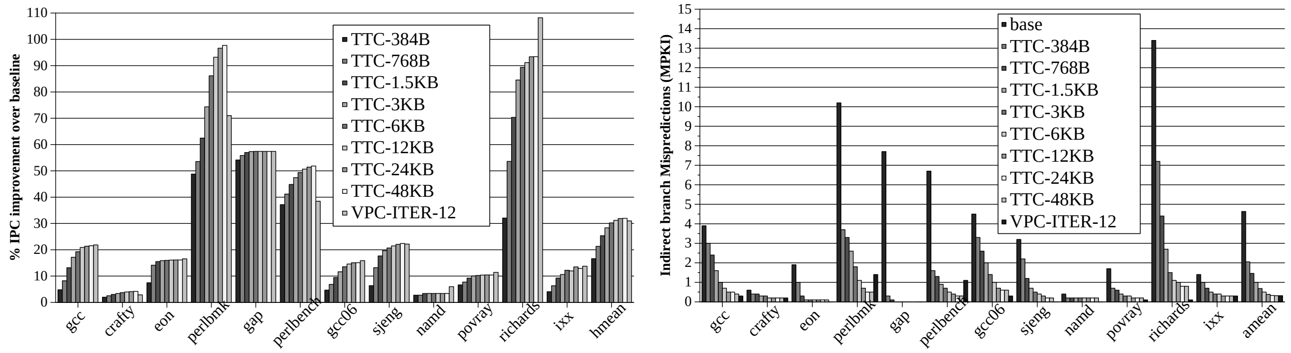

**Figure 12. Performance of VPC prediction vs. TTC on a processor with an O-GEHL conditional branch predictor: IPC (left), MPKI (right)**

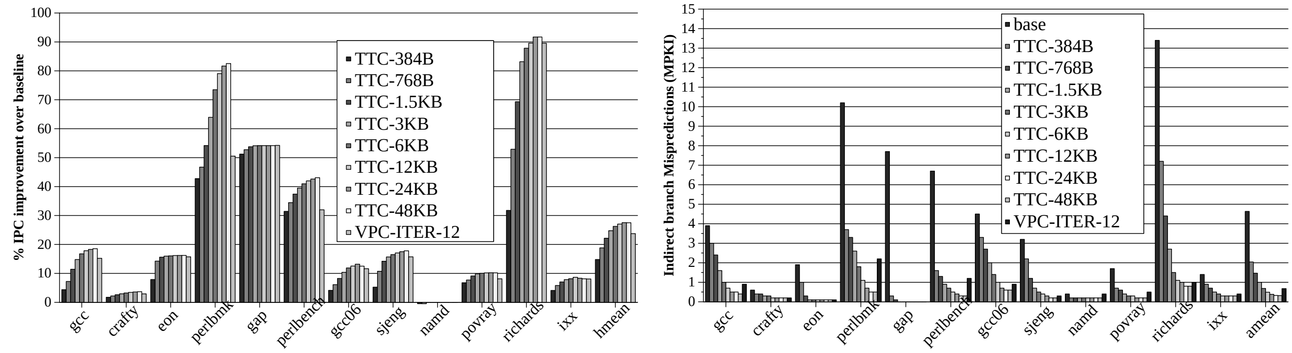

**Figure 13. Performance of VPC prediction vs. TTC on a processor with a gshare conditional branch predictor: IPC (left), MPKI (right)**

**5.6.2. Different BTB sizes** We evaluated VPC prediction with different BTB sizes: 512, 1024, 2048, 4096 (base), 8192 and 16384 entries.[14] As Table 8 shows, even with smaller BTB sizes VPC prediction still provides significant performance improvements. With a very small (512-entry) BTB, VPC prediction improves performance by 18.5%. With a very large (16K-entry) BTB, the performance improvement is 27.1%. Even though VPC prediction slightly increases the BTB miss rate for conditional branch instructions (due to the contention it introduces by storing multiple targets per indirect branch in the BTB), the negative impact of this is easily offset by the positive performance impact the large reductions in the indirect branch misprediction rate (as Table 8 shows). We conclude that VPC prediction becomes more effective as BTB size increases, but it is still effective with a small BTB.

**Table 8. Effect of different BTB sizes**

| BTB entries | Baseline | | | VPC prediction | | |
|---|---|---|---|---|---|---|
| | indirect MPKI | cond br. BTB miss (%) | IPC | indirect MPKI | cond br. BTB miss (%) | IPC Δ |
| 512 | 4.81 | 8.2 | 1.16 | **1.31** | 8.4 | **18.5%** |
| 1K | 4.65 | 2.9 | 1.25 | **0.95** | 3.0 | **21.7%** |
| 2K | 4.64 | 0.7 | 1.28 | **0.78** | 0.7 | **23.8%** |
| 4K | 4.63 | 0.1 | 1.29 | **0.52** | 0.1 | **26.7%** |
| 8K | 4.63 | 0.01 | 1.29 | **0.46** | 0.01 | **27.0%** |
| 16K | 4.63 | 0.006 | 1.29 | **0.45** | 0.006 | **27.1%** |

**5.6.3. VPC Prediction on a Less Aggressive Processor** Figure 14 shows the performance of VPC and TTC predictors on a less aggressive baseline processor that has a 20-stage pipeline, 4-wide fetch/issue/retire rate, 128-entry

[14] Note that many modern processors have large BTBs: AMD Athlon (2K-entry) [1], Intel Pentium 4 (4K-entry) [22], IBM z990 (8K-entry) [49].

instruction window, 16KB perceptron branch predictor, 1K-entry BTB, and 200-cycle memory latency. Since the less aggressive processor incurs a smaller penalty for a branch misprediction, improved indirect branch handling provides smaller performance improvements than in the baseline processor. However, VPC prediction still improves performance by 17.6%.

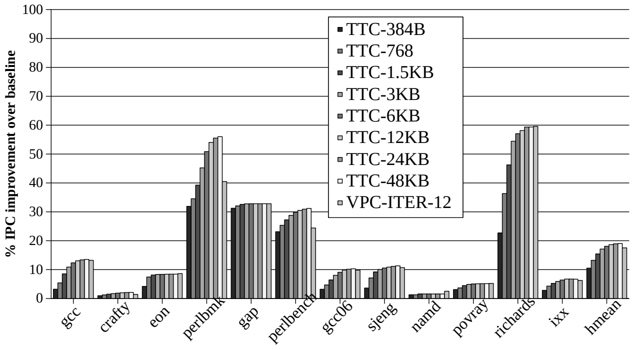

Figure 14. VPC prediction vs. TTC on a less aggressive processor

### 5.7. Effect of VPC Prediction on Power and Energy Consumption

Figure 15 shows the impact of VPC prediction and TTC predictors of different sizes on maximum processor power, overall energy consumption, energy-delay product of the processor, and the energy consumption of the branch prediction logic (which includes conditional/indirect predictors and the BTB). We used the Wattch infrastructure [4] to measure power/energy consumption, faithfully modeling every processing structure and the additional accesses to the branch predictors. The power model is based on 100nm technology and a 4GHz processor.

On average, VPC prediction reduces the overall energy consumption by 19%, which is higher than the energy reduction provided by the most energy-efficient TTC predictor (12KB). The energy reduction is due to the reduced pipeline flushes and thus reduced amount of time the processor spends fetching and executing wrong-path instructions. Furthermore, VPC prediction reduces the energy delay product (EDP) by 42%, which is also higher than the EDP reduction provided by the most energy-efficient TTC predictor. VPC prediction improves EDP significantly because it improves performance while at the same time reducing energy consumption.

VPC prediction does not significantly increase the maximum power consumption of the processor whereas even a 3KB TTC predictor results in a 0.3% increase in maximum power consumption due to its additional hardware overhead. Note that relatively large TTC predictors significantly increase not only the complexity but also the energy consumption of the branch prediction unit. We conclude that VPC prediction is an energy-efficient way of improving processor performance without significantly increasing the complexity of the processor frontend and the overall processor power consumption.

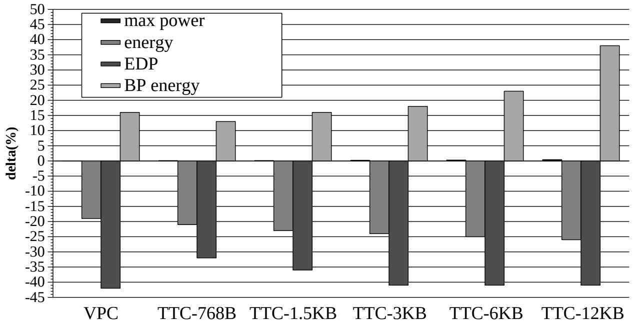

**Figure 15. Effect of VPC prediction on energy/power consumption**

### 5.8. Performance of VPC Prediction on Server Applications

We also evaluated the VPC predictor with commercial on-line transaction processing (OLTP) applications [21]. Each OLTP trace is collected from an IBM System 390 zSeries machine [25] for 22M instructions. Unlike the SPEC CPU benchmarks, OLTP applications have a much higher number of static indirect branches (OLTP1:7601, OLTP2:7991, and OLTP3:15733) and very high indirect branch misprediction rates.[15] The VPC predictor (MAX_ITER=10) reduces the indirect branch misprediction rate by 28%, from 12.2 MPKI to 8.7 MPKI. The VPC predictor performs better than a 12KB TTC predictor on all applications and almost as well as a 24KB TTC on oltp2. Hence, we conclude that the VPC predictor is also very effective in large-scale server applications.

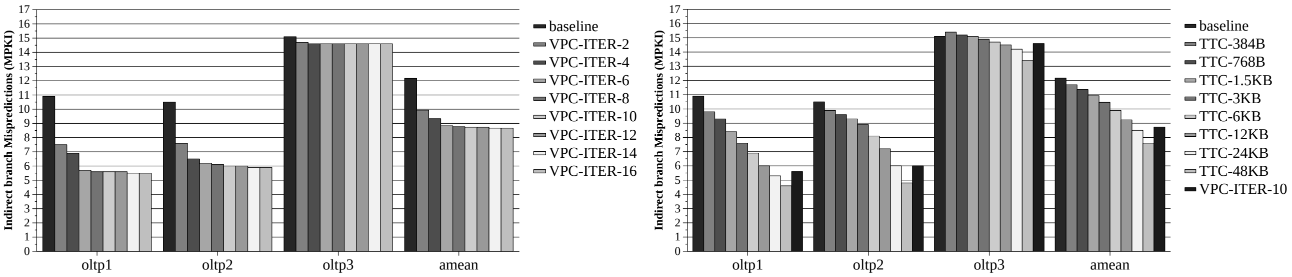

**Figure 16. MPKI of VPC prediction on OLTP Applications: effect of MAX_ITER (left) and vs. TTC predictor (right)**

## 6. VPC Prediction and Compiler-Based Devirtualization

*Devirtualization* is the substitution of an indirect method call with direct method calls in object-oriented languages [10, 24, 20, 5, 29]. Ishizaki et al. [29] classify the devirtualization techniques into *guarded devirtualization* and *direct devirtualization*.

**Guarded devirtualization:** Figure 17a shows an example virtual function call in the C++ language. In the example, depending on the actual type of `Shape s`, different `area` functions are called at run-time. However, even though there could be many different shapes in the program, if the types of shapes are mostly either an instance of the

[15] System 390 ISA has both unconditional indirect and conditional indirect branch instructions. For this experiment, we only consider unconditional indirect branches and use a 16K-entry 4-way BTB in the baseline processor. Since server applications have very large indirect branch working sets, BTBs of processors designed for use in server systems are also large [49].

`Rectangle` class or the `Circle` class at run-time, the compiler can convert the indirect call to multiple guarded direct calls [20, 17, 5] as shown in Figure 17(b). This compiler optimization is called Receiver Class Prediction Optimization (*RCPO*) and the compiler can perform RCPO based on profiling.

```
Shape* s = ... ;
a = s->area(); //  an indirect call
```

(a) A virtual function call in C++

```
Shape * s = ...;
if (s->class == Rectangle)   // a cond. br at PC: X
    a = Rectangle::area();   // a direct call
else if (s->class == Circle) // a cond. br at PC: Y
    a = Circle::area();      // a direct call
else
    a = s->area();     // an indirect call at PC: Z
```

(b) Devirtualized form of the above virtual function call

**Figure 17. A virtual function call and its devirtualized form**

The benefits of this optimization are: (1) It enables other compiler optimizations. The compiler could inline the direct function calls or perform interprocedural analysis [17]. Removing function calls also reduces the register save/restore overhead. (2) The processor can predict the virtual function call using a conditional branch predictor, which usually has higher accuracy than an indirect branch predictor [5]. However, not all indirect calls can be converted to multiple conditional branches. In order to perform RCPO, the following conditions need to be fulfilled [17, 5]:

1. The number of frequent target addresses from a caller site should be small (1-2).
2. The majority of target addresses should be similar across input sets.
3. The target addresses must be available at compile-time.

**Direct devirtualization:** Direct devirtualization converts an indirect call into a single unconditional direct call if the compiler can prove that there is only one possible target for the indirect call. Hence, direct devirtualization does not require a guard before the direct call, but requires whole-program analysis to make sure there is only one possible target. This approach enables code optimizations that would otherwise be hindered by the indirect call. However, this approach cannot be used statically if the language supports dynamic class loading, like Java. Dynamic recompilation can overcome this limitation, but it requires an expensive mechanism called on-stack replacement [29].

## 6.1. Limitations of Compiler-Based Devirtualization

**6.1.1. Need for Static Analysis or Accurate Profiling** The application of devirtualization to large commercial software bases is limited by the cost and overhead of the static analysis or profiling required to guide the method call transformation. Devirtualization based on static analysis requires type analysis, which in turn requires whole program analysis [29], and unsafe languages like C++ also require pointer alias analysis. Note that these analyses need to be conservative in order to guarantee correct program semantics. Guarded devirtualization usually requires accurate profile information, which may be very difficult to obtain for large applications. Due to the limited applicability of static devirtualization, [29] reports only an average 40% reduction in the number of virtual method calls on a set of Java benchmarks, with the combined application of aggressive guarded and direct devirtualization techniques.

**6.1.2. Impact on Code Size and Branch Mispredictions** Guarded devirtualization can sometimes reduce performance since (1) it increases the static code size by converting a single indirect branch instruction into multiple guard test instructions and direct calls; (2) it could replace one possibly mispredicted indirect call with multiple conditional branch mispredictions, if the guard tests become hard-to-predict branches [50].

**6.1.3. Lack of Adaptivity to Run-Time Input-Set and Phase Behavior** The most frequently-taken targets chosen for devirtualization can be based on profiling, which averages the whole execution of the program for one particular input set. However, the most frequently-taken targets can be different across different input sets. Furthermore, the most frequently-taken targets can change during different phases of the program. Additionally, dynamic linking and dynamic class loading can introduce new targets at runtime. Compiler-based devirtualization cannot adapt to these changes in program behavior because the most frequent targets of a method call are determined statically and encoded in the binary.

Due to these limitations, many state-of-the-art compilers either do not implement any form of devirtualization (e.g. GCC 4.0 [18][16]) or they implement a limited form of direct devirtualization that converts only provably-monomorphic virtual function calls into direct function calls (e.g. the Bartok compiler [50, 41] or the .NET Runtime [42]).

### 6.2. VPC Prediction vs. Compiler-Based Devirtualization

VPC prediction is essentially a *dynamic devirtualization* mechanism used for indirect branch prediction purposes. However, VPC's devirtualization is visible only to the branch prediction structures. VPC has the following advantages over compiler-based devirtualization:

1. As it is a hardware mechanism, it can be applied to *any indirect branch* without requiring any static analysis/guarantees or profiling.

2. Adaptivity: Unlike compiler-based devirtualization, the dynamic training algorithms allow the VPC predictor to adapt to changes in the most frequently-taken targets or even to new targets introduced by dynamic linking or dynamic class loading.

3. Because virtual conditional branches are visible only to the branch predictor, VPC prediction does not increase

[16] GCC only implements a form of devirtualization based on class hierarchy analysis in the *ipa-branch* experimental branch, but not in the main branch [43].

the code size, nor does it possibly convert a single indirect branch misprediction into multiple conditional branch mispredictions.

On the other hand, the main advantage of compiler-based devirtualization over VPC prediction is that it enables compile-time code optimizations. However, as we show in the next section, the two techniques can be used in combination and VPC prediction provides performance benefits on top of compiler-based devirtualization.

**Table 9. The number of static and dynamic indirect branches in BASE and RCPO binaries**

| | gcc | crafty | eon | perlbmk | gap | perlbench | gcc06 | sjeng | namd | povray |
|---|---|---|---|---|---|---|---|---|---|---|
| Static BASE | 987 | 356 | 1857 | 864 | 1640 | 1283 | 1557 | 369 | 678 | 1035 |
| Static RCPO | 984 | 358 | 1854 | 764 | 1709 | 930 | 1293 | 369 | 333 | 578 |
| Dynamic BASE (M) | 144 | 174 | 628 | 1041 | 2824 | 8185 | 304 | 10130 | 7 | 8228 |
| Dynamic RCPO (M) | 94 | 119 | 619 | 1005 | 2030 | 1136 | 202 | 10132 | 4 | 7392 |

### 6.3. Performance of VPC Prediction on Binaries Optimized with Compiler-Based Devirtualization

A compiler that performs devirtualization reduces the number of indirect branches and therefore reduces the potential performance improvement of VPC prediction. This section evaluates the effectiveness of VPC prediction on binaries that are optimized using aggressive profile-guided optimizations, which include RCPO. ICC [26] performs a form of RCPO [46] when value-profiling feedback is enabled, along with other profile-based optimizations.[17]

Table 9 shows the number of static/dynamic indirect branches in the *BASE* and *RCPO* binaries run with the full reference input set. *BASE* binaries are compiled with the `-O3` option. *RCPO* binaries are compiled with all profile-guided optimizations, including RCPO.[18] Table 9 shows that RCPO binaries reduce the number of static/dynamic indirect branches by up to 51%/86%.

Figure 18 shows the performance impact of VPC prediction on RCPO binaries. Even though RCPO binaries have fewer indirect branches, VPC prediction still reduces indirect branch mispredictions by 80% on average, improving performance by 11.5% over a BTB-based predictor. Figure 19 shows the performance comparison of VPC prediction with a tagged target cache on RCPO binaries. The performance of VPC is better than a tagged predictor of 48KB (for eon, namd, povray), and equivalent to a tagged predictor of 24KB (for gap), of 12KB (for gcc), of 3KB (for perlbmk, gcc06, and sjeng), of 1.5KB (for crafty), and 768B (for perlbench). Hence, a VPC predictor provides the performance of a large and more complicated tagged target cache predictor even when the RCPO optimization is used by the compiler.

## 7. Evaluation of VPC Prediction on Object-Oriented Java Applications

This section evaluates VPC prediction using a set of modern object-oriented Java applications, the full set of DaCapo benchmarks [3]. Our goal is to demonstrate the benefits of VPC prediction on real object-oriented applications

[17] Since it is not possible to selectively enable only RCPO in ICC, we could not isolate the impact of RCPO on performance. Hence, we only present the effect of VPC prediction on binaries optimized with RCPO.

[18] RCPO binaries were compiled in two passes with ICC: the first pass is a profiling run with the train input set (`-prof_gen` switch), and the second pass optimizes the binaries based on the profile (we use the `-prof_use` switch, which enables all profile-guided optimizations).

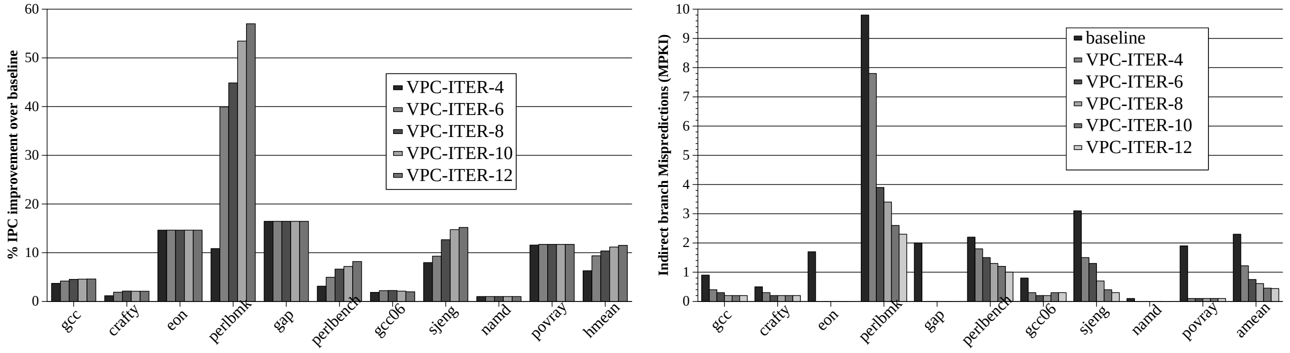

**Figure 18. Performance of VPC prediction on RCPO binaries: IPC (left) and MPKI (right)**

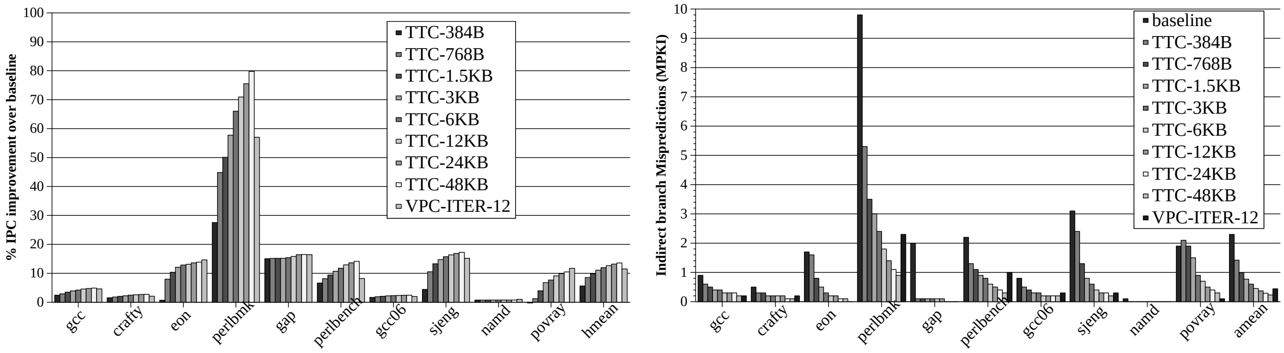

**Figure 19. VPC prediction vs. tagged target cache on RCPO binaries: IPC (left) and MPKI (right)**

and to analyze the differences in the behavior of VPC prediction on object-oriented Java programs versus on traditional C/C++ programs (which were evaluated in Section 5).

### 7.1. Methodology

We have built an iDNA-based [2] cycle-accurate x86 simulator to evaluate VPC prediction on Java applications. iDNA [2] is a dynamic binary instrumentation tool similar to Pin [37], but capable of tracing Java virtual machines. The DaCapo benchmarks are run with Sun J2SE 1.4.2_15 JRE on Windows Vista. Each benchmark is run for 200 million x86 instructions with the small input set. The parameters of our baseline processor are the same as those we used to evaluate VPC prediction on C/C++ applications as shown in 2.[19]

Table 10 shows the characteristics of the examined Java programs on the baseline processor. Compared to the evaluated C/C++ programs, the evaluated Java programs have significantly higher number of static and dynamic indirect branches and indirect branch misprediction rates (also see Table 4). We found that this difference is due to the object-oriented nature of the Java programs, which contain a large number of virtual functions, and the behavior of the Java Virtual Machine, which uses a large number of indirect branches in its interpretation and dynamic translation phases [14]. As a result, the potential performance improvement possible with perfect indirect branch prediction is

[19] We use a BTB size of 8K entries to evaluate Java applications since they are very branch-intensive. However, we also evaluate other BTB sizes in Section 7.5.1.

significantly higher in the evaluated Java applications (73.1%) than in the evaluated C/C++ applications (32.5%).

**Table 10. Characteristics of the evaluated Java applications:** baseline IPC (BASE IPC), potential IPC improvement with perfect indirect branch prediction (PIBP IPC $\Delta$), static number of indirect branches (Static IB), dynamic number of indirect branches (Dyn. IB), indirect branch prediction accuracy (IBP Acc), indirect branch mispredictions per kilo instructions (IB MPKI), conditional branch mispredictions per kilo instructions (CB MPKI), and average number of dynamic targets.

| | antlr | bloat | chart | eclipse | fop | hsqldb | jython | luindex | lusearch | pmd | xalan | AVG |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BASE IPC | 0.98 | 0.92 | 0.77 | 1.20 | 0.79 | 1.21 | 1.20 | 1.15 | 1.12 | 1.01 | 0.77 | 0.98 |
| PIBP IPC $\Delta$ | 80.3% | 71.2% | 48.4% | 56.9% | 130.9% | 57.5% | 57.8% | 60.1% | 65.3% | 70.1% | 114.4% | 73.1% |
| Static IB | 800 | 628 | 917 | 1579 | 1155 | 2533 | 1548 | 1587 | 1585 | 944 | 795 | - |
| Dyn. IB | 4917K | 5390K | 4834K | 3523K | 7112K | 3054K | 3565K | 3744K | 4054K | 4557K | 6923K | - |
| IBP Acc (%) | 49.3 | 54.1 | 51.8 | 52.0 | 44.7 | 61.2 | 51.9 | 51.4 | 51.8 | 49.8 | 44.6 | 51.2 |
| IB MPKI | 12.5 | 12.4 | 11.6 | 8.5 | 19.7 | 8.3 | 8.6 | 9.1 | 9.8 | 11.4 | 19.2 | 11.9 |
| CB MPKI | 2.5 | 2.2 | 2.4 | 4.5 | 3.1 | 3.1 | 4.4 | 4.6 | 4.3 | 3.9 | 3.9 | 3.5 |
| 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 | 39.7 | - |

## 7.2. Dynamic Target Distribution of Java Applications

Figure 20 shows the distribution of the number of dynamic targets for executed indirect branches. Unlike C/C++ programs evaluated in Section 5.1, only 14% of executed indirect branches have a single target and 53% of them have more than 20 targets (Recall that 51% of the indirect branches in the evaluated C/C++ programs had more than 5 targets). On average, 76% of the dynamic indirect branches in the evaluated Java benchmarks have more than 5 targets, in contrast to the 51% in the evaluated indirect-branch intensive C/C++ programs. Only in hsqldb more than 20% of the dynamic indirect branches have only one target, which are easily predictable with a simple BTB-based indirect branch predictor. The high number of targets explains why the evaluated Java programs have higher indirect branch misprediction rates than the evaluated C/C++ programs.

We found that there are two major reasons for the high number of dynamic targets in the Java applications: 1) The evaluated Java applications are written in object-oriented style. Therefore, they include many *polymorphic virtual function calls*, i.e. virtual function calls that are overridden by many derived classes, whose overridden forms are exercised at run time. 2) The Java virtual machine itself uses a significant number of indirect jumps with many targets in its interpretation routines, as shown in previous work on virtual machines [14].

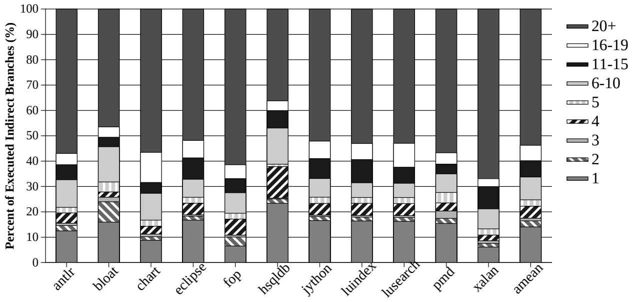

**Figure 20. Distribution of the number of dynamic targets across executed indirect branches in the Java programs**

### 7.3. Performance of VPC Prediction on Java Applications

Figure 21 (left) shows the performance improvement of VPC prediction over the baseline BTB-based predictor when MAX_ITER is varied from 2 to 16. Figure 21 (right) shows the indirect branch misprediction rate (MPKI) in the baseline and with VPC prediction. Similarly to the results for C/C++ benchmarks, a MAX_ITER value of 12 provides the highest performance improvement. *All* of the 11 Java applications experience more than 10% performance improvement with VPC prediction and 10 of the 11 applications experience more than 15% performance improvement. This shows that the benefits of VPC prediction are very consistent across different object-oriented Java applications. On average, VPC prediction provides 21.9% performance improvement in the Java applications.

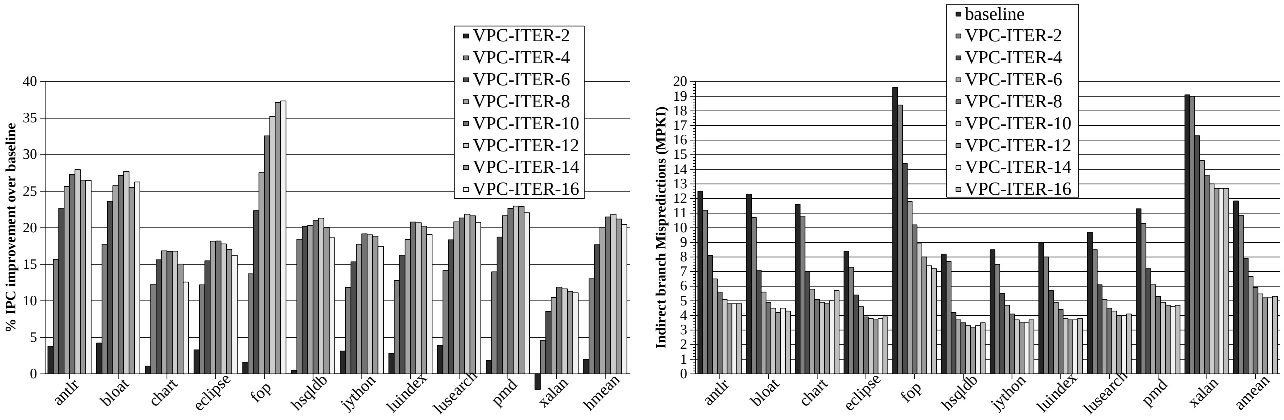

**Figure 21. Performance of VPC prediction on Java applications: IPC improvement (left), indirect branch MPKI (right)**

**7.3.1. Analysis** Since the majority of indirect branches have more than 10 targets, as MAX_ITER increases, the indirect branch MPKI decreases (from 11.9 to 5.2), until MAX_ITER equals 12. The most significant drop in MPKI (from 10.9 to 7.9) happens when MAX_ITER is increased from 2 to 4 (meaning VPC prediction can store four different targets for a branch rather than two). However, when MAX_ITER is greater than 12, MPKI starts increasing in most of the evaluated Java applications (unlike in C/C++ applications where MPKI continues to decrease). This is due to the pressure extra virtual branches exert on the BTB: as Java applications have a large number of indirect branches with a large number of dynamically-exercised targets, more targets contend for the BTB space with higher values of MAX_ITER. As a result, BTB miss rate for virtual branches increases and the prediction accuracy of VPC prediction decreases. When the MPKI increase is combined with the additional iteration cycles introduced for some predictions by higher MAX_ITER values, the performance improvement of VPC prediction drops from 21.9% (for MAX_ITER=12) to 20.4% (for MAX_ITER=16).

Even though VPC prediction significantly reduces the misprediction rate from 11.9 to 5.2 MPKI in Java applications, a significant number of mispredictions still remain. This is in contrast to the results we obtained for C/C++

applications where VPC prediction was able to eliminate 89% of all mispredictions (down to 0.63 MPKI). Hence, indirect branches in Java applications are more difficult to predict. Therefore, other techniques like dynamic predication [31] might be needed to complement VPC prediction to further reduce the impact of indirect branches on Java application performance.

Figure 22 shows the distribution of the number of iterations needed to generate a correct target prediction. On average 44.8% of the correct predictions occur in the first iteration (i.e. zero idle cycles) and 78.7% of the correct predictions occur within four iterations. Hence, most correct predictions are performed quickly resulting in few idle cycles during which the fetch engine stalls. Note that the number of iterations (cycles) it takes to make a correct prediction is higher for Java applications than for C/C++ applications because indirect branches in Java applications have a significantly higher number of dynamically-exercised targets per indirect branch.

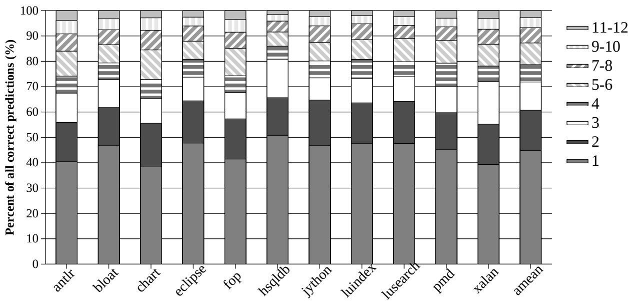

**Figure 22. Distribution of the number of iterations (for correct predictions) in the Java programs (MAX_ITER=12)**

### 7.4. VPC Prediction versus Other Indirect Branch Predictors on Java Applications

Figure 23 compares the performance and MPKI of VPC prediction with the tagged target cache (TTC) predictor [9]. On average, VPC prediction provides performance improvement equivalent to that provided by a 3-6 KB TTC predictor (similarly to the results for C/C++ applications).[20]

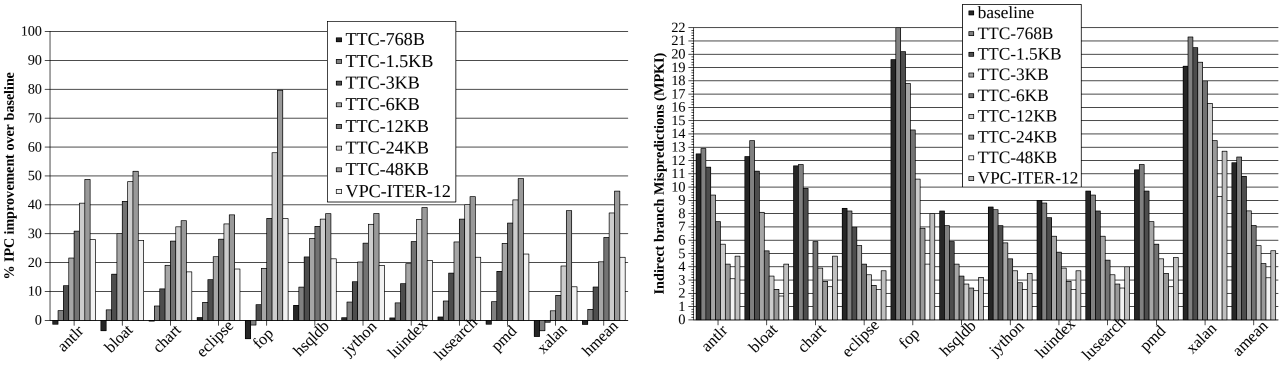

**Figure 23. Performance of VPC prediction vs. tagged target cache: IPC (left), MPKI (right)**

[20] In the examined Java applications, increasing the size of the TTC predictor up to 48KB continues providing large performance improvements, whereas doing so results in very little return in performance for C/C++ applications. A larger TTC predictor is better able to accommodate the large indirect branch target working set of Java applications whereas a small TTC predictor is good enough to accommodate the small target working set of C/C++ applications. Hence the difference in the effect of TTC size on performance between Java versus C/C++ applications.

Figure 24 compares the performance and MPKI of VPC prediction with the cascaded predictor. On average, VPC prediction provides the performance provided by a 5.5-11KB cascaded predictor. Because the number of static indirect branches is very high in Java applications, a small cascaded predictor (cascaded-704B) performs significantly worse than the baseline BTB-based predictor. This behavior is not seen in C/C++ benchmarks because those benchmarks have much fewer indirect branches with smaller number of targets that do not cause significant contention in the tables of a small cascaded predictor. However, even though there are many static indirect branches in the examined Java applications, VPC predictor still provides significant performance improvements equaling those of large cascaded predictors, without requiring extra storage for indirect branch targets.

Note that the size of the TTC or cascaded predictor that provides the same performance as VPC prediction is smaller for Java applications than for C/C++ applications. In other words, TTC and cascaded predictors are relatively more effective in Java than C/C++ applications. This is because of the large indirect branch and target working set size of Java applications, which can better utilize the extra target storage space provided by specialized indirect branch predictors.

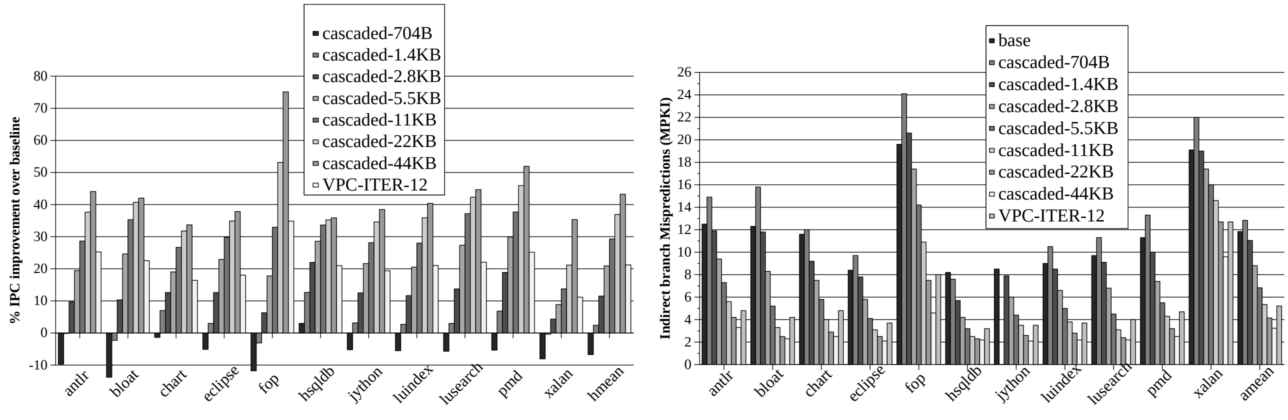

**Figure 24. Performance of VPC prediction vs. cascaded predictor on Java applications: IPC (left), MPKI (right)**

### 7.5. Effect of Microarchitecture Parameters on VPC Prediction Performance on Java Applications

**7.5.1. Effect of BTB Size** Table 11 shows the effect of the baseline BTB size on VPC prediction performance on Java applications. Similarly to what we observed for C/C++ applications, VPC prediction provides higher performance improvements as BTB size increases. However, with smaller BTB sizes, VPC prediction's performance improvement is smaller on Java applications than on C/C++ applications. For example, with a 512-entry BTB, VPC prediction improves the performance of Java applications by 6.3% whereas it improves the performance of C/C++ applications by 18.5% (as was shown in Table 8). As Java applications have very large indirect branch and target address working sets, VPC prediction results in a larger contention (i.e., conflict misses) in the BTB in these applications than in C/C++ applications, thereby delivering a smaller performance improvement. Even so, the performance improvement provided by VPC prediction with very small BTB sizes is significant for Java applications. We conclude that VPC prediction is very effective on Java applications for a wide variety of BTB sizes.

Table 11. Effect of different BTB sizes in Java applications

| BTB entries | Baseline | | | VPC prediction | | |
|---|---|---|---|---|---|---|
| | indirect MPKI | Cond. Br BTB Miss (%) | IPC | indirect MPKI | Cond. Br BTB Miss (%) | IPC $\Delta$ |
| 512 | 13.10 | 8.9 | 0.87 | **10.17** | 9.5 | **6.3%** |
| 1K | 12.36 | 3.7 | 0.94 | **8.31** | 4.8 | **11.1%** |
| 2K | 12.05 | 2.1 | 0.97 | **6.77** | 2.3 | **17.5%** |
| 4K | 11.92 | 0.9 | 0.97 | **5.99** | 1.0 | **19.6%** |
| 8K | 11.94 | 0.3 | 0.98 | **5.21** | 0.3 | **21.9%** |

**7.5.2. Effect of a Less Aggressive Processor** Figure 25 shows the performance of VPC and TTC predictors on a less aggressive baseline processor that has a 20-stage pipeline, 4-wide fetch/issue/retire rate, 128-entry instruction window, 16KB perceptron branch predictor, 4K-entry BTB, and 200-cycle memory latency. Similarly to our observation for C/C++ applications, since the less aggressive processor incurs a smaller penalty for a branch misprediction, improved indirect branch handling provides smaller performance improvements than in the baseline processor. However, VPC prediction still improves performance of Java applications by 11.1% on a less aggressive processor. In fact, all Java applications except xalan experience very close to or more than 10% performance improvement with VPC prediction. This is different from what we have seen for C/C++ applications on the less aggressive processor: some applications saw very large performance improvements with VPC prediction whereas others saw very small (less than 5% as shown in Figure 14). Thus, we conclude that VPC prediction's performance improvements are very consistent across the Java applications on both aggressive and less aggressive baseline processors.

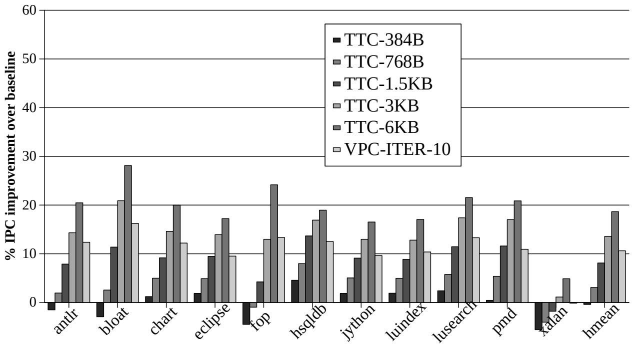

Figure 25. VPC prediction vs. TTC on a less aggressive processor

### 7.6. Effect of VPC Prediction on Power and Energy Consumption of Java Applications

Figure 26 shows the impact of VPC prediction and TTC/cascaded predictors of different sizes on maximum processor power, overall energy consumption, energy-delay product of the processor, and the energy consumption of the branch prediction logic. On average, VPC prediction reduces the overall energy consumption by 22%, and energy delay product (EDP) by 36%. Similarly to what we observed for C/C++ applications, VPC prediction provides larger reductions in energy consumption on Java applications than the most energy-efficient TTC predictor (12KB) as well as the most energy-efficient cascaded predictor (11KB). Moreover, VPC prediction does not significantly increase maximum power consumption (less than 0.1%) whereas a 12KB TTC predictor and an 11KB cascaded predictor result in respectively 2.1% and 2.2% increase in power consumption due to the extra storage and prediction structures they require. We conclude that VPC prediction is an energy- and power-efficient indirect branch handling technique that provides significant performance improvements in object-oriented Java applications without significantly increasing the energy consumption or complexity of the processor front-end.

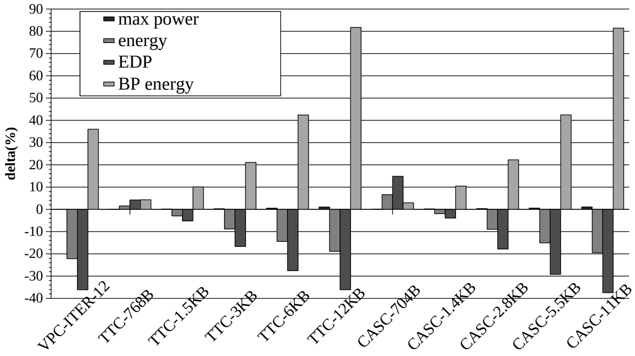

**Figure 26. Effect of VPC prediction on energy/power consumption on Java applications**

To provide more insight into the reduction in energy consumption and EDP, Figure 27 shows the percentage change in pipeline flushes, fetched instructions, and executed instructions due to VPC prediction and TTC/cascaded predictors. VPC prediction reduces the number of pipeline flushes by 30.1%, which results in a 47% reduction in the number of fetched instructions and a 23.4% reduction in the number of executed instructions. Hence, VPC prediction reduces energy consumption significantly due to the large reduction in the number of fetched/executed instructions. Notice that even though a 12KB TTC predictor provides a larger reduction in pipeline flushes, it is less energy-efficient than the VPC predictor due to the significant extra hardware it requires.

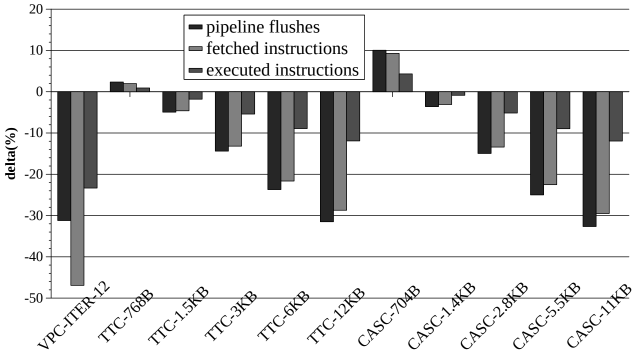

**Figure 27. Pipeline flushes, and fetched/executed instructions on Java applications**

## 8. Other Related Work

We have already discussed related work on indirect branch prediction in Section 2.2. Sections 5, 6 and 7 provide extensive comparisons of VPC prediction with three of the previously proposed indirect branch predictors, finding that VPC prediction, without requiring significant hardware, provides the performance benefits provided by other predictors of much larger size. Here, we briefly discuss other related work in handling indirect branches.

We [31] recently proposed handling hard-to-predict indirect branches using dynamic predication [35]. In this technique, if the target address of an indirect branch is found to be hard to predict, the processor selects two (or more) *likely* targets and follows the control-flow paths after all of the targets by dynamically predicating the instructions on each path. When the indirect branch is resolved, instructions on the control-flow paths corresponding to the incorrect targets turn into NOPs. Unlike VPC prediction, dynamic predication of indirect branches requires compiler support, new instructions in the instruction set architecture, and significant hardware support for dynamic predication (as described in [35]). However, the two approaches can be combined and used together: dynamic predication can be a promising approach to reduce the performance impact of indirect branches that are hard to predict with VPC prediction.

Roth et al. [45] proposed dependence-based pre-computation, which pre-computes targets for future virtual function calls as soon as an object reference is created. This technique avoids a misprediction if the result of the computation is correct and ready to be used when the future instance of the virtual function call is fetched. However, it requires a dedicated and costly precomputation engine. In contrast, VPC prediction has two advantages: 1) it does not require any pre-computation logic, 2) it is generally applicable to any indirect branch rather than only for virtual function calls.

Pure software approaches have been proposed specifically for mitigating the performance impact due to virtual function calls. These approaches include the method cache in Smalltalk-80 [10], polymorphic inline caches [23] and type feedback/devirtualization [24, 29]. As we show in Section 6, the benefit of devirtualization is limited by its lack of adaptivity. We compare and contrast VPC prediction with compiler-based devirtualization extensively in Section 6.

Finally, Ertl and Gregg [14] proposed code replication and superinstructions to improve indirect branch prediction accuracy on virtual machine interpreters. In contrast to this scheme, VPC prediction is not specific to any platform and is applicable to any indirect branch.

## 9. Conclusion

This paper proposed and evaluated the VPC prediction paradigm. The key idea of VPC prediction is to treat an indirect branch instruction as multiple “virtual” conditional branch instructions for prediction purposes in the microarchitecture. As such, VPC prediction enables the use of existing conditional branch prediction structures to predict the targets of indirect branches without requiring any extra structures specialized for storing indirect branch targets. Our evaluation shows that VPC prediction, without requiring complicated structures, achieves the performance provided by other indirect branch predictors that require significant extra storage and complexity. On a set of indirect branch intensive C/C++ applications and modern object-oriented Java applications, VPC prediction provides respectively 26.7% and 21.9% performance improvement, while also reducing energy consumption significantly.

We believe the performance impact of VPC prediction will further increase in future applications that will be written in object-oriented programming languages and that will make heavy use of polymorphism since those languages were shown to result in significantly more indirect branch mispredictions than traditional C/Fortran-style languages. By making available to indirect branches the rich, accurate, highly-optimized, and continuously-improving hardware used to predict conditional branches, VPC prediction can serve as an enabler encouraging programmers (especially those concerned with the performance of their code) to use object-oriented programming styles, thereby improving the quality and ease of software development.

## References

[1] Advanced Micro Devices, Inc. *AMD Athlon$^{(TM)}$ XP Processor Model 10 Data Sheet*, Feb. 2003.

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

[3] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, 2006.

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

[5] B. Calder and D. Grunwald. Reducing indirect function call overhead in C++ programs. In *POPL-21*, 1994.

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

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

[8] P.-Y. Chang, M. Evers, and Y. N. Patt. Improving branch prediction accuracy by reducing pattern history table interference. In *PACT*, 1996.

[9] P.-Y. Chang, E. Hao, and Y. N. Patt. Target prediction for indirect jumps. In *ISCA-24*, 1997.

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

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

[12] K. Driesen and U. Hölzle. The cascaded predictor: Economical and adaptive branch target prediction. In *MICRO-31*, 1998.

[13] K. Driesen and U. Hölzle. Multi-stage cascaded prediction. In *European Conference on Parallel Processing*, 1999.

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

[15] M. Evers, S. J. Patel, R. S. Chappell, and Y. N. Patt. An analysis of correlation and predictability: What makes two-level branch predictors work. In *ISCA-25*, 1998.

[16] The GAP Group. *GAP System for Computational Discrete Algebra*. `http://www.gap-system.org/`.

[17] C. Garrett, J. Dean, D. Grove, and C. Chambers. Measurement and application of dynamic receiver class distributions. Technical Report UW-CS 94-03-05, University of Washington, Mar. 1994.

[18] GCC-4.0. GNU compiler collection. http://gcc.gnu.org/.

[19] 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.

[20] D. Grove, J. Dean, C. Garrett, and C. Chambers. Profile-guided receiver class prediction. In *OOPSLA-10*, 1995.

[21] A. Hartstein and T. R. Puzak. The optimum pipeline depth for a microprocessor. In *ISCA-29*, 2002.

[22] 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. Q1 2001 Issue.

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

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

[25] IBM Corporation. IBM zSeries mainframe servers. `http://www.ibm.com/systems/z/`.

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

[27] Intel Corporation. Intel Core Duo Processor T2500. `http://processorfinder.intel.com/Details.aspx?sSpec=SL8VT`.

[28] Intel Corporation. *Intel VTune Performance Analyzers*. `http://www.intel.com/vtune/`.

[29] 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.

[30] D. A. Jiménez and C. Lin. Dynamic branch prediction with perceptrons. In *HPCA-7*, 2001.

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

[32] D. Kaeli and P. Emma. Branch history table predictions of moving target branches due to subroutine returns. In *ISCA-18*, 1991.

[33] J. Kalamatianos and D. R. Kaeli. Predicting indirect branches via data compression. In *MICRO-31*, 1998.

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

[35] 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.

[36] J. K. F. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, Jan. 1984.

[37] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[38] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[39] T. McDonald. Microprocessor with branch target address cache update queue. U.S. Patent Number 7,165,168, 2007.

[40] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Western Research Laboratory, June 1993.

[41] Microsoft Research. Bartok compiler. `http://research.microsoft.com/act/`.

[42] V. Morrison. Digging into interface calls in the .NET Framework: Stub-based dispatch. http://blogs.msdn.com/vancem/archive/2006/03/13/550529.aspx.

[43] D. Novillo, Mar. 2007. Personal communication.

[44] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *MICRO-37*, 2004.

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

[46] D. Sehr, Nov. 2006. Personal communication.

[47] A. Seznec. Analysis of the O-GEometric History Length branch predictor. In *ISCA-32*, 2005.

[48] A. Seznec and P. Michaud. A case for (partially) TAgged GEometric history length branch prediction. *Journal of Instruction-Level Parallelism (JILP)*, 8, Feb. 2006.

[49] T. J. Slegel, E. Pfeffer, and J. A. Magee. The IBM eServer z990 microprocessor. *IBM Journal of Research and Development*, 48(3/4):295–309, May/July 2004.

[50] D. Tarditi, Nov. 2006. Personal communication.

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

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

[53] T.-Y. Yeh, D. Marr, and Y. N. Patt. Increasing the instruction fetch rate via multiple branch prediction and branch address cache. In *ICS*, 1993.