Virtual Program Counter (VPC) Prediction: Very Low Cost Indirect Branch Prediction Using Conditional Branch Prediction Hardware

Hyesoon Kim, *Member*, *IEEE*, José A. Joao, *Student Member*, *IEEE*, Onur Mutlu, *Member*, *IEEE*, Chang Joo Lee, *Student Member*, *IEEE*, Yale N. Patt, *Fellow*, *IEEE*, and Robert Cohn

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 use the existing conditional branch prediction hardware to predict indirect branch targets, avoiding the need for a separate storage structure. Our comprehensive evaluation shows that VPC prediction improves average performance by 26.7 percent and reduces average energy consumption by 19 percent compared to a commonly used branch target buffer based predictor on 12 indirect branch intensive C/C++ applications. Moreover, VPC prediction improves the average performance of the full set of object-oriented Java DaCapo applications by 21.9 percent, while reducing their average energy consumption by 22 percent. 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.

Index Terms—Indirect branch prediction, virtual functions, devirtualization, object-oriented languages, Java.

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 [8], 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 runtime 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 [7]. 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 [50].

Fig. 1 shows the number and fraction of indirect branch mispredictions per 1000 retired instructions (MPKI) in different Windows applications run on an Intel Core Duo T2500 processor [26] which includes a specialized indirect branch predictor [20]. The data are collected with hardware performance counters using VTune [27]. In the examined Windows applications, on average 28 percent of the branch

- *H. Kim is with the College of Computing, Georgia Institute of Technology, 266 Ferst Drive, Atlanta, GA 30332-0765. E-mail: hyesoon@cc.gatech.edu.*
- *J.A. Joao, C.J. Lee, and Y.N. Patt are with the Department of Electrical and Computer Engineering, University of Texas at Austin, 1 University Station C0803, Austin, TX 78712-0240. E-mail: {joao, cjlee, patt}@ece.utexas.edu.*
- *O. Mutlu is with the Department of Electrical and Computer Engineering, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213-3890. E-mail: onur@cmu.edu.*
- *R. Cohn is with Intel Corporation, M/S: HD2-300, 77 Reed Road, Hudson, MA 01749. E-mail: robert.s.cohn@intel.com.*

Manuscript received 3 Oct. 2007; revised 2 June 2008; accepted 18 Sept. 2008; published online 19 Dec. 2008.
Recommended for acceptance by A. George.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2007-10-0498.
Digital Object Identifier no. 10.1109/TC.2008.227.

1. In the rest of this paper, an "indirect branch" refers to a nonreturn 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].

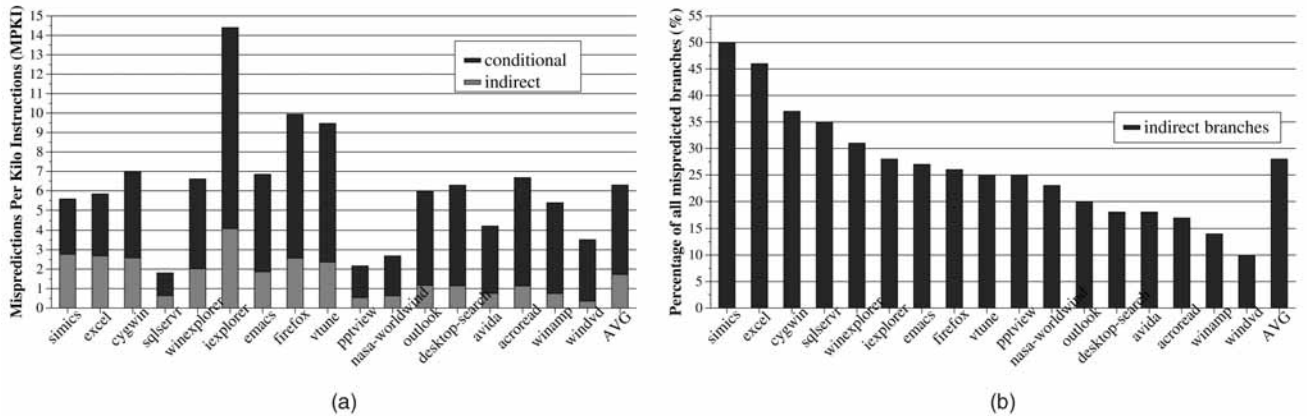


Fig. 1. Indirect branch mispredictions in Windows applications: (a) MPKI and (b) percent of mispredictions due to indirect branches.

mispredictions are due to indirect branches. In two programs, Virtutech Simics [39] 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 [10], [12], [33], [13], [14], [47] require large hardware resources to store the target addresses of indirect branches. For example, a 1,024-entry gshare conditional branch predictor [41] requires only 2,048 bits but a 1,024-entry gshare-like indirect branch predictor (tagged target cache [10]) needs at least 2,048 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 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]. The BTB implicitly—and usually inaccurately—assumes that the indirect branch will jump to the same target address it jumped to in its previous execution [10], [33].[3] To our knowledge, only Intel Pentium M and AMD Barcelona implement specialized hardware to help the prediction of indirect branches [20], [2], demonstrating that hardware designers are increasingly concerned with the performance impact of indirect branches. However, as we showed in Fig. 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 if it were multiple conditional branches for prediction purposes only. That is, the code has the single indirect branch. But the hardware branch predictor treats it in the hardware for prediction purposes only as a sequence of conditional branch instructions. These conditional branch instructions do not exist in software. They are part of the hardware branch prediction mechanism. Ergo, we call them *virtual branches*.

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 the *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 of the virtual branch. If the predictor returns "taken," the target address provided by the BTB is predicted as the target of the indirect branch. If the predictor returns "not-taken," the processor moves on to the next virtual branch in the sequence.[4]

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. Our results in Section 5.2 show that the number of iterations needed to generate a correct target prediction is actually small: 45 percent of the correct predictions occur in the first virtual branch and 81 percent of the correct predictions occur within the first three virtual branches.

The VPC prediction algorithm is inspired by a compiler optimization, called *receiver class prediction optimization (RCPO)* [11], [24], [21], [6] or *devirtualization* [28]. This optimization statically converts an indirect branch to

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

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 percent) because the target addresses of many indirect branches alternate rather than stay stable for long periods of time [10], [33].

4. Since we are using the outcomes of the existing conditional branch predictor to predict indirect branches, we refer to the two outcomes of the predictor as "taken" and "not-taken." However, what we mean by "not-taken" when we are processing indirect branches is not "take the fall through path in the actual code" as is the case for *real* conditional branches. What we mean is "process the next virtual branch in our sequence of virtual branches that collectively represent the indirect branch." In other words, update the branch history, and make another pass at the conditional branch predictor.

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 [28]. Our proposed VPC prediction mechanism 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 percent/21.9 percent and decreasing energy consumption by 19 percent/22 percent, 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

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 [46]. The technique we introduce in this paper utilizes history information, so we focus on history-based indirect branch predictors.

```
1: // Set up the array of function pointers (i.e. jump table)
2: EvTab[T_INT] = Eval_INT;  EvTab[T_VAR] = Eval_VAR;
3: EvTab[T_SUM] = Eval_SUM;
4: // ...
5:
6: // EVAL evaluates an expression by calling the function
7: //  corresponding to the type of the expression
8: //  using the EvTab[] array of function pointers
9:
10: #define EVAL(hd) ((*EvTab[TYPE(hd)])((hd))) /*INDIRECT*/
11:
12: TypHandle  Eval_LISTELEMENT ( TypHandle hdSel ) {
13:     hdPos = EVAL( hdSel );
14:     // evaluate the index of the list element
15:     // check if index is valid and within bounds
16:     // if within bounds, access the list
17:     // at the given index and return the element
18: }
```

Fig. 2. An indirect branch example from GAP.

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 [10]. 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 [16].

2.1.1 A Source Code Example

The example in Fig. 2 shows an indirect branch from the GAP program [17] 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 Fig. 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 for L1(n), L1(n+1), and L2(n) by calling the Eval_VAR, Eval_SUM, and Eval_VAR functions, respectively. 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 [37] used the 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 objects of different classes and switch statements with many "case" targets that are exercised at runtime. Therefore, the BTB-based predictor has low (about 50 percent) prediction accuracy [37], [10], [12], [33].

Chang et al. [10] 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 [41] 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 [12] and by combining multiple indirect branch predictors using a cascaded predictor [13], [14]. 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 [14] concluded that a three-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. Similarly to a cascaded predictor, the prediction comes from the highest order table that can predict. 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 [47] 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 (four or seven 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 multistage 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.[5] Moreover, many of the previously proposed indirect branch predictors are themselves complicated [13], [14], [33], [47], 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 conditional branches, called *virtual branches*. A "virtual branch" is conceptually a conditional branch that is visible only to the processor's branch prediction structures. As such, it is different from a "real" conditional branch; it does not affect program behavior, it is not part of the program binary, and it is only used by the VPC predictor. Each virtual branch is predicted in sequence using the existing conditional branch prediction hardware, which consists of the direction predictor and the BTB (Fig. 3). If the direction predictor predicts the virtual branch as not-taken, the VPC predictor moves on to predict the next virtual branch in the sequence. If the direction predictor predicts the virtual branch as 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.

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 direction*

5. 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.

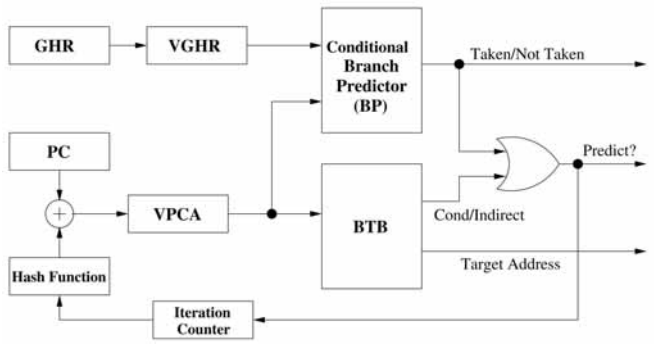


Fig. 3. High-level conceptual overview of the VPC predictor.

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 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. 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.[6] Note that 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. 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).[7] If the prediction process stops without predicting a target, the processor stalls until the indirect branch is resolved. Our results in Section 5.2 show that 81 percent of the correct predictions happen in the first three iterations.

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.

3.2.1 Prediction Example

Figs. 4a and 4b show an example virtual function call and the corresponding simplified assembly code with an indirect branch. Fig. 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 Fig. 4c are shown to demonstrate VPC prediction conceptually. We assume, for this example, that MAX_ITER is 3, so there are only three virtual conditional branches.

Algorithm 1. VPC prediction algorithm

```
iter ← 1
VPCA ← PC
VGHR ← GHR
done ← FALSE
while (!done) do
  pred_target ← access_BTB(VPCA)
  pred_dir ← access_conditional_BP(VPCA, VGHR)
  if (pred_target and(pred_dir = TAKEN)) then
    next_PC ← pred_target
    done ← TRUE
  else if (!pred_target or(iter ≥ MAX_ITER)) then
    STALL ← TRUE
    done ← TRUE
  end if
  VPCA ← Hash(PC, iter)
  VGHR ← Left-Shift(VGHR)
  iter++
end while
```

Table 1 demonstrates five possible cases when the indirect branch in Fig. 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,

6. 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) codesign 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 [41], [9] can also be used to reduce aliasing due to VPC prediction. However, our experimental results in [35, Section 5.5] and in Section 7.5 show that the negative effect of VPC prediction on the BTB miss rate and conditional branch misprediction rate is tolerable.

7. The VPC predictor can continue iterating the prediction process even if there is a 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.

```
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)

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

TABLE 1
Possible VPC Predictor States and Outcomes When Branch in Fig. 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

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.[8]

8. 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.

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.

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

```
iter ← 1
VPCA ← PC
VGHR ← 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 ← Hash(PC, iter)
  VGHR ← Left-Shift(VGHR)
  iter++
end while
```

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.

Algorithm 3. VPC training algorithm when the branch target is mispredicted. Inputs: *PC*, *GHR*, *CORRECT_TARGET*

```
iter ← 1
VPCA ← PC
VGHR ← GHR
found_correct_target ← FALSE
while ((iter ≤ MAX_ITER) and (found_correct_target = FALSE)) do
  pred_target ← access_BTB(VPCA)
  if (pred_target = CORRECT_TARGET) then
    update_conditional_BP(VPCA, VGHR, TAKEN)
    update_replacement_BTB(VPCA)
    found_correct_target ← TRUE
  else if (pred_target) then
    update_conditional_BP(VPCA, VGHR, NOT-TAKEN)
  end if
  VPCA ← Hash(PC, iter)
  VGHR ← Left-Shift(VGHR)
  iter++
  end while

/* no-target case */
if (found_correct_target = FALSE) then
  VPCA ← VPCA corresponding to the virtual branch
  with a BTB-Miss or Least-frequently-used target among
  all virtual branches
  VGHR ← 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
```

To distinguish between *wrong-target* and *no-target* cases, the training logic accesses the BTB for each virtual branch (as shown in Algorithm 3).[9] 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.)

9. 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.

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 [51]. 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. We found that the performance benefit of supporting multiple predictions per cycle is not significant (see [35, Section 5.4]).

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).

TABLE 2
Baseline Processor Configuration

Front End	64KB, 2-way, 2-cycle I-cache; fetch ends at the first predicted-taken branch; fetch up to 3 conditional branches or 1 indirect branch
Branch Predictors	64KB (64-bit history, 1021-entry) perceptron branch predictor [29]; 4K-entry, 4-way BTB with pseudo-LFU replacement; 64-entry return address stack; min. branch misprediction penalty is 30 cycles
Execution Core	8-wide fetch/issue/execute/retire; 512-entry ROB; 384 physical registers; 128-entry LD-ST queue; 4-cycle pipelined wake-up and selection logic; scheduling window is partitioned into 8 sub-windows of 64 entries each
On-chip Caches	L1 D-cache: 64KB, 4-way, 2-cycle, 2 ld/st ports; L2 unified cache: 1MB, 8-way, 8 banks, 10-cycle latency; All caches use LRU replacement and have 64B line size
Buses and Memory	300-cycle minimum memory latency; 32 memory banks; 32B-wide core-to-memory bus at 4:1 frequency ratio
Prefetcher	Stream prefetcher with 32 streams and 16 cache line prefetch distance (lookahead) [49]

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 [40].)

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 an O/S kernel [50]

4 Experimental Methodology

We use a Pin-based [38] 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 [37].

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

We use Pinpoints [45] 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) [25] with the -O3 optimization level.

TABLE 4
Characteristics of the Evaluated Benchmarks

	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 Δ	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

Language and type of the benchmark (Lang/Type), baseline IPC (BASE IPC), potential IPC improvement with perfect indirect branch prediction (PIBP IPC Δ), 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.

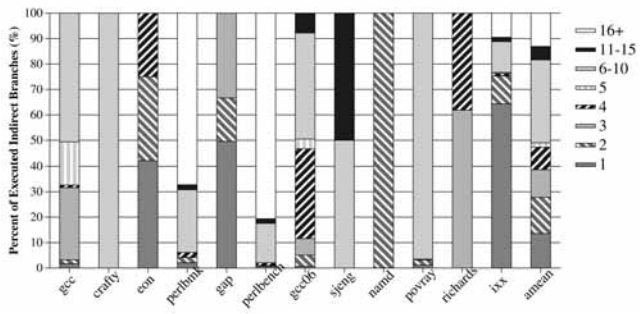


Fig. 5. Distribution of the number of dynamic targets across executed indirect branches.

5 Results

We have provided an extensive performance characterization of VPC prediction on C/C++ applications in our previous paper [35]. In particular, in [35], we provided the characteristics of indirect branch targets of C/C++ applications, performance comparisons to other indirect branch predictors, sensitivity of VPC prediction to microarchitecture parameters, and performance of VPC prediction on server applications. In this paper, after briefly summarizing the performance of VPC prediction, we focus our attention to the training options for VPC prediction (Section 5.3), power/energy consumption analysis (Section 5.4), and the evaluation of VPC prediction on Java applications (Section 7).

5.1 Dynamic Target Distribution

We first briefly analyze the behavior of indirect branch targets in our benchmark set. Fig. 5 shows the distribution of the number of dynamic targets for executed indirect branches. In eon, gap, and ixx, more than 40 percent 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 percent), crafty (100 percent), perlbmk (94 percent), perlbench (98 percent), sjeng (100 percent), and povray (97 percent), over 50 percent of the dynamic indirect branches have more than five targets. On average, 51 percent of the dynamic indirect branches in the evaluated benchmarks have more than five targets.

5.2 Performance of VPC Prediction

Fig. 6a shows the performance improvement of VPC prediction over the baseline BTB-based predictor when MAX_ITER is varied from 2 to 16. Fig. 6b shows the indirect branch MPKI in the baseline and with VPC prediction. In eon, gap, and namd, where over 60 percent of all executed indirect branches have at most two unique targets (as shown in Fig. 5), VPC prediction with MAX_ITER=2 eliminates almost all indirect branch mispredictions. Almost all indirect branches in richards have three or four different targets. Therefore, when the VPC predictor can hold four 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 percent 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 and 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 percent 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 [35].

Fig. 7 shows the distribution of the number of iterations needed to generate a correct target prediction. On average, 44.6 percent of the correct predictions occur in the first iteration (i.e., zero idle cycles) and 81 percent of the correct predictions occur within three iterations. Only in perlbmk

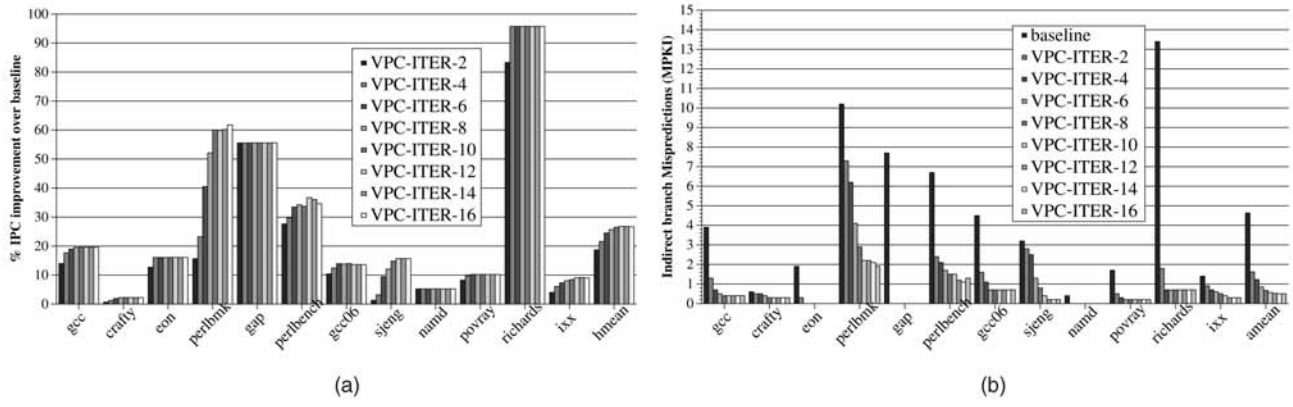


Fig. 6. Performance of VPC prediction: (a) IPC improvement and (b) indirect branch MPKI.

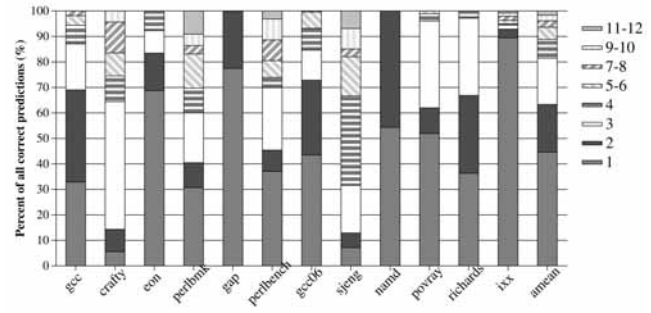


Fig. 7. Distribution of the number of iterations (for correct predictions) (MAX_ITER=12).

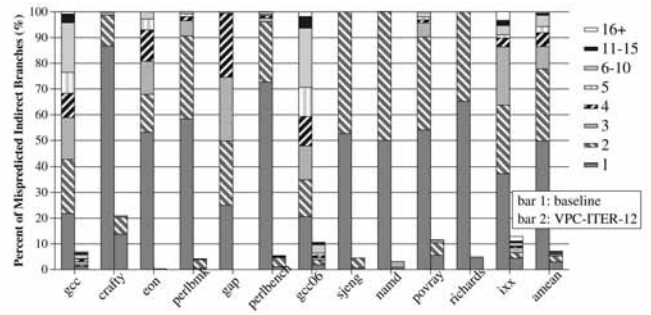


Fig. 8. Fraction of mispredictions caused by the Nth most mispredicted static indirect branch. The portions of each bar labeled "1" and "2" are the fraction of all mispredictions caused by, respectively, the "most" and "second most" mispredicted indirect branch, and so on. The data are normalized to the number of mispredictions in the baseline.

and sjeng, more than 30 percent of all correct predictions require at least five iterations. Hence, most correct predictions are performed quickly resulting in few idle cycles during which the fetch engine stalls.

Fig. 8 provides insight into the performance improvement of VPC prediction by showing the distribution of mispredictions per highly mispredicted static indirect branches using both the baseline BTB predictor and the VPC predictor. The left bar for each benchmark shows the distribution of mispredictions per the Nth most mispredicted static indirect branch, sorted by number of mispredictions, using the baseline BTB-based predictor. The portions of each bar labeled "1" and "2" are the fraction of all mispredictions caused by, respectively, the "most" and "second most" mispredicted indirect branch, and so on. The most mispredicted static indirect branch causes on average approximately 50 percent of all indirect branch mispredictions. (This fraction varies between 21 percent and 87 percent depending on the benchmark.) The data show that only a few indirect branches are responsible for the majority of the mispredictions. The second set of bars in Fig. 8 shows the distribution of mispredictions for *the same static indirect branches* using the VPC predictor, normalized to the number of mispredictions with the BTB-based predictor. The VPC predictor significantly reduces or eliminates the mispredictions across the board, i.e., for almost all of the highly mispredicted indirect branches. We conclude that VPC prediction is effective at reducing misprediction rate across a large number of different indirect branches.

Even though VPC prediction is very effective at reducing the indirect branch misprediction rate, it does not completely

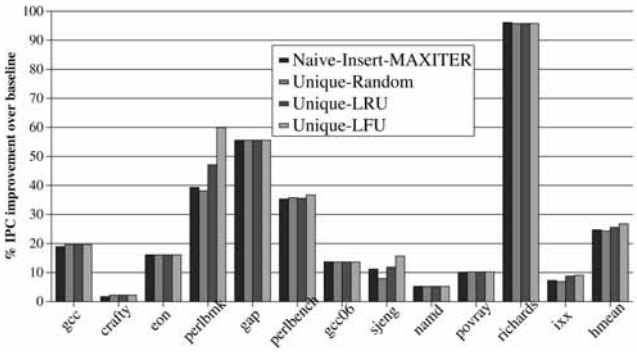


Fig. 9. Performance impact of different VPC training schemes.

eliminate indirect branch mispredictions. This is due to two reasons. The first reason is algorithmic: VPC prediction cannot correctly predict a target address when the target is not correlated with branch history in the way our prediction and training algorithms can capture. The second reason is due to resource contention: the contention and interference in BTB entries and conditional branch direction predictor entries between both conditional and indirect branches as well as different targets of indirect branches can lead to mispredictions. We analyzed the effects of such contention and interference in [35].

5.3 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.

Fig. 9 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 that it increases the redundancy of target addresses in the BTB (hence, it reduces 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*

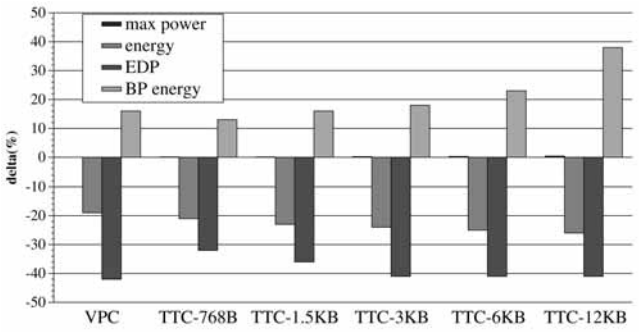


Fig. 10. Effect of VPC on energy/power consumption.

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 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 Fig. 9, 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 Fig. 5 and [35]), 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 percent and 2 percent better than *Unique-LRU* and *Unique-Random, respectively*). 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. Hence, we have chosen the *Unique-LFU* scheme as our default VPC training scheme.

5.4 Effect on Power and Energy Consumption

Fig. 10 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 [5] to measure power/energy consumption, faithfully modeling every processing structure and the additional accesses to the branch predictors. The power model is based on 100 nm technology and a 4 GHz processor.

On average, VPC prediction reduces the overall energy consumption by 19 percent, which is higher than the energy reduction provided by the most energy-efficient TTC predictor (12 KB). 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 percent, 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 3 KB TTC predictor results in a 0.3 percent 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 front end and the overall processor power consumption.

6 VPC Prediction and Compiler-Based Devirtualization

Devirtualization is the substitution of an indirect method call with direct method calls in object-oriented languages [11], [24], [21], [6], [28]. Ishizaki et al. [28] classify the devirtualization techniques into *guarded devirtualization* and *direct devirtualization*.

Guarded devirtualization. Fig. 11a 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 runtime. However, even though there could be many different shapes in the program, if the types of shapes are mostly either an instance of the `Rectangle` class or the `Circle` class at runtime, the compiler can convert the indirect call to multiple guarded direct calls [21], [18], [6], as shown in Fig. 11b. This compiler optimization is called Receiver Class Prediction Optimization (*RCPO*) and the compiler can perform RCPO based on profiling.

The benefits of this optimization are: 1) it enables other compiler optimizations. The compiler could inline the direct function calls or perform interprocedural analysis [18]. 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

```
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

Fig. 11. A virtual function call and its devirtualized form.

usually has higher accuracy than an indirect branch predictor [6]. However, not all indirect calls can be converted to multiple conditional branches. In order to perform RCPO, the following conditions need to be fulfilled [18], [6]:

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 that there is only one possible target. This approach enables code optimizations that would otherwise be hindered by the indirect call. However, this approach cannot usually 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 [28].

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 [28], 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, Ishizaki et al. [28] report only an average 40 percent 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 and 2) it could replace one possibly mispredicted indirect call with multiple conditional branch mispredictions, if the guard tests become hard-to-predict branches [48].

6.1.3 Lack of Adaptivity

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 [19][10]) 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 [48], [42] or the .NET Runtime [43]).

6.2 VPC versus 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 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 showed in our previous paper [35], the two techniques can be used in combination and VPC prediction provides performance benefits on top of compiler-based devirtualization. In particular, using VPC prediction on binaries that are already optimized with compiler-based devirtualization improves performance by 11.5 percent (see [35, Section 6.3] for a detailed analysis).

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 [4]. Our goal is to demonstrate the benefits of VPC prediction on real object-oriented applications 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 briefly in Section 5 and extensively in [35]).

7.1 Methodology

We have built an iDNA-based [3] cycle-accurate x86 simulator to evaluate VPC prediction on Java applications.

10. GCC only implements a form of devirtualization based on class hierarchy analysis in the *ipa-branch* experimental branch, but not in the main branch [44].

TABLE 5
Characteristics of the Evaluated Java Applications (See Table 4 for Explanation of Abbreviations)

	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 Δ	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	-

iDNA [3] is a dynamic binary instrumentation tool similar to Pin [38], 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 Table 2.[11]

Table 5 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 [15]. As a result, the potential performance improvement possible with perfect indirect branch prediction is significantly higher in the evaluated Java applications (73.1 percent) than in the evaluated C/C++ applications (32.5 percent).

7.2 Dynamic Target Distribution of Java Programs

Fig. 12 shows the distribution of the number of dynamic targets for executed indirect branches. Unlike C/C++ programs evaluated in Section 5.1, only 14 percent of executed indirect branches have a single target and 53 percent of them have more than 20 targets (recall that 51 percent of the indirect branches in the evaluated C/C++ programs had more than five targets). On average, 76 percent of the dynamic indirect branches in the evaluated Java benchmarks have more than five targets, in contrast to the 51 percent in the evaluated indirect-branch intensive C/C++ programs. Hsqldb is the only benchmark where more than 20 percent of the dynamic indirect branches have only one target, and these monomorphic branches 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 runtime. 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 [15].

11. We use a BTB size of 8,192 entries to evaluate Java applications since they are very branch intensive. However, we also evaluate other BTB sizes in Section 7.5.1.

7.3 Performance of VPC Prediction on Java Programs

Fig. 13a shows the performance improvement of VPC prediction over the baseline BTB-based predictor when MAX_ITER is varied from 2 to 16. Fig. 13b 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 percent performance improvement with VPC prediction and 10 of the 11 applications experience more than 15 percent 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 percent performance improvement in the Java applications.

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

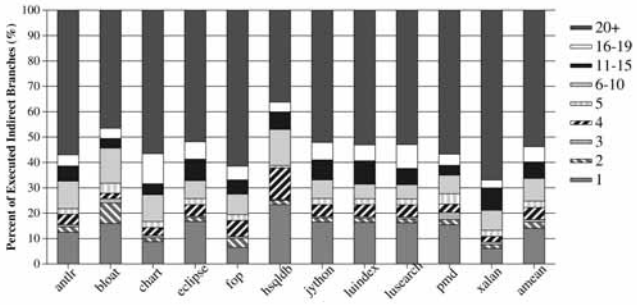


Fig. 12. Distribution of the number of dynamic targets across executed indirect branches in the Java programs.

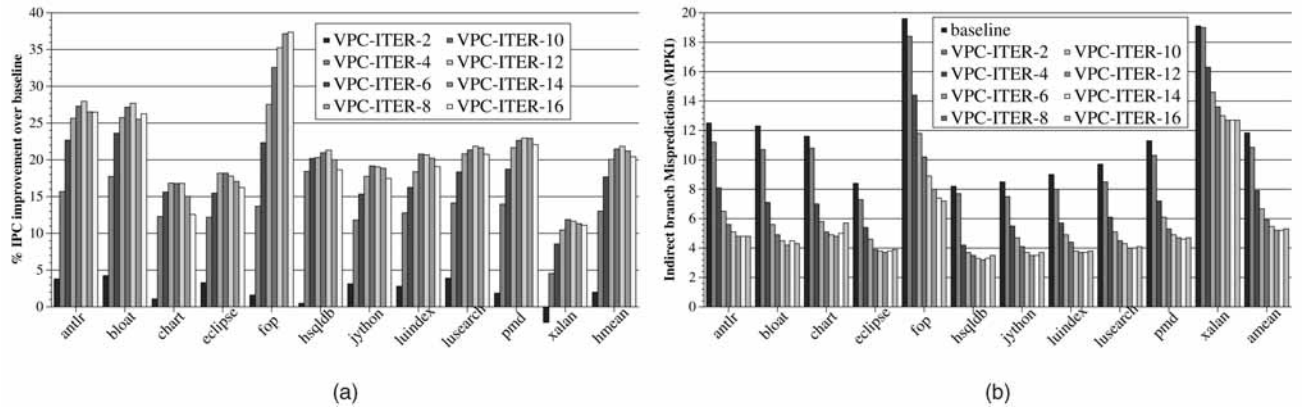


Fig. 13. Performance of VPC prediction on Java applications: (a) IPC improvement and (b) indirect branch MPKI.

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 percent (for MAX_ITER=12) to 20.4 percent (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 percent 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], [30] might be needed to complement VPC prediction to further reduce the impact of indirect branches on Java application performance.

Fig. 14 shows the distribution of the number of iterations needed to generate a correct target prediction. On average, 44.8 percent of the correct predictions occur in the first iteration (i.e., zero idle cycles) and 78.7 percent 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.

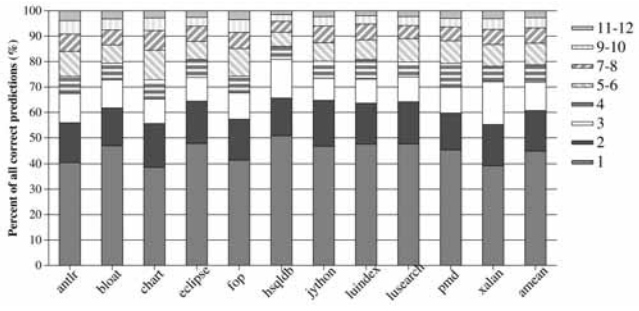


Fig. 14. Distribution of the number of iterations (for correct predictions) in the Java programs (MAX_ITER=12).

7.4 VPC Prediction versus Other Indirect Branch Predictors on Java Applications

Fig. 15 compares the performance of VPC prediction with the tagged target cache (TTC) predictor [10]. 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 [35]).[12]

Fig. 16 compares the performance of VPC prediction with the cascaded predictor. On average, VPC prediction provides the performance provided by a 5.5-11 KB cascaded predictor. Because the number of static indirect branches is very high in Java applications, a small cascaded predictor (cascaded-704 B) performs significantly worse than the baseline BTB-based predictor. This behavior is not seen in C/C++ benchmarks because these 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 [35]. 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.

12. In the examined Java applications, increasing the size of the TTC predictor up to 48 KB 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.

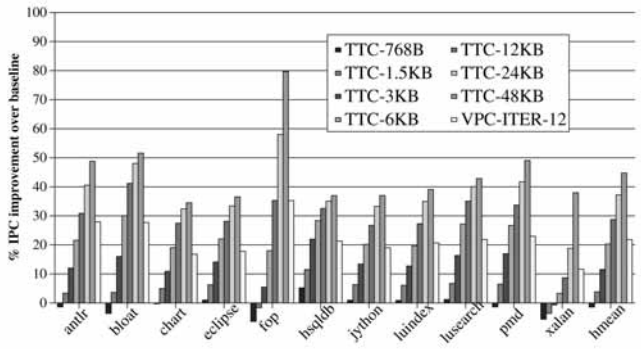


Fig. 15. Performance of VPC prediction versus tagged target cache.

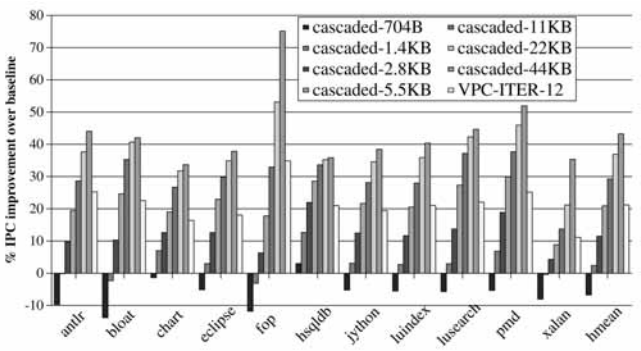


Fig. 16. Performance of VPC prediction versus cascaded predictor.

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

7.5.1 Effect of BTB Size

Table 6 shows the effect of the baseline BTB size on VPC prediction performance on Java applications. Similarly to what we observed for C/C++ applications [35], 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 percent whereas it improves the performance of C/C++ applications by 18.5 percent [35]. 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.

7.5.2 Effect of a Less Aggressive Processor

Fig. 17 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, 16 KB perceptron branch predictor, 4K-entry BTB, and 200-cycle memory latency. Similarly to our observation for C/C++ applications [35], 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 percent on a less aggressive processor. In fact, all Java applications except xalan experience very close to or more than 10 percent 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. Thus, we conclude that VPC prediction's performance improvements are very consistent across the Java applications on both aggressive and less aggressive baseline processors.

TABLE 6
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 Δ
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%**

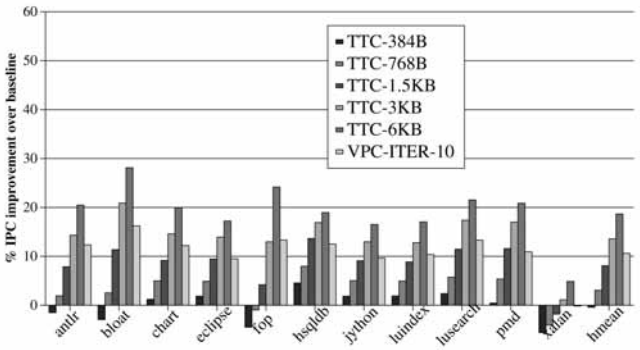


Fig. 17. VPC prediction versus TTC on a less aggressive processor.

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

Fig. 18 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 percent, and energy-delay product (EDP) by 36 percent. Similarly to what we observed for C/C++ applications in Section 5.4, VPC prediction provides larger reductions in energy consumption on Java applications than the most energy-efficient TTC predictor (12 KB) as well as the most energy-efficient cascaded predictor (11 KB). Moreover, VPC prediction does not significantly increase maximum power consumption (less than 0.1 percent) whereas a 12 KB TTC predictor and an 11 KB cascaded predictor result in, respectively, 2.1 percent and 2.2 percent 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

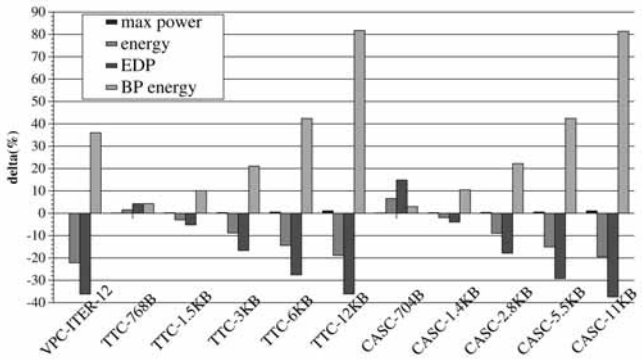


Fig. 18. Energy/power consumption on Java programs.

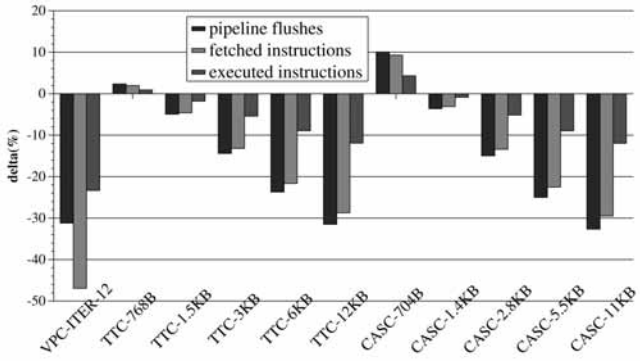


Fig. 19. Pipeline flushes and fetched/executed instructions.

technique that provides significant performance improvements in object-oriented Java applications without significantly increasing the energy consumption or complexity of the processor front end.

To provide more insight into the reduction in energy consumption and EDP, Fig. 19 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 percent, which results in a 47 percent reduction in the number of fetched instructions and a 23.4 percent 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. Note that even though a 12 KB 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.

8 Other Related Work

We have already discussed related work on indirect branch prediction in Section 2.2. [35], and 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], [30] recently proposed handling hard-to-predict indirect branches using dynamic predication [36]. 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 [36]). 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. [46] proposed dependence-based precomputation, which precomputes 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 and 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 [11], polymorphic inline caches [23], and type feedback/devirtualization [24], [28]. 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 [15] 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, respectively, provides 26.7 percent and 21.9 percent 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 these 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.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their feedback. They also thank Thomas Puzak, Joel Emer, Paul Racunas, John Pieper, Robert Cox, David Tarditi, Veynu Narasiman, Rustam Miftakhutdinov, Jared Stark, Santhosh Srinath, Thomas Moscibroda, Bradford Beckmann, and the other members of the HPS research group for their comments and suggestions. They gratefully acknowledge the support of the Cockrell Foundation, Intel Corporation, Microsoft Research, and the Advanced Technology Program of the Texas Higher Education Coordinating Board. This paper is an extended version of [35].

REFERENCES

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

[2] Advanced Micro Devices, Inc., *Software Optimization Guide for AMD Family 10h Processors,* Apr. 2008.

[3] 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," *Proc. Second Int'l Conf. Virtual Execution Environments (VEE '06),* 2006.

[4] 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," *Proc. 21st Ann. ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '06),* 2006.

[5] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," *Proc. 27th Ann. Int'l Symp. Computer Architecture (ISCA '00),* 2000.

[6] B. Calder and D. Grunwald, "Reducing Indirect Function Call Overhead in C++ Programs," *Proc. 21st ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL '94),* 1994.

[7] B. Calder, D. Grunwald, and B. Zorn, "Quantifying Behavioral Differences between C and C++ Programs," *J. Programming Languages,* vol. 2, no. 4, pp. 323-351, 1995.

[8] L. Cardelli and P. Wegner, "On Understanding Types, Data Abstraction, and Polymorphism," *ACM Computing Surveys,* vol. 17, no. 4, pp. 471-523, Dec. 1985.

[9] P.-Y. Chang, M. Evers, and Y.N. Patt, "Improving Branch Prediction Accuracy by Reducing Pattern History Table Interference," *Proc. Conf. Parallel Architectures and Compilation Techniques (PACT '96),* 1996.

[10] P.-Y. Chang, E. Hao, and Y.N. Patt, "Target Prediction for Indirect Jumps," *Proc. 24th Ann. Int'l Symp. Computer Architecture (ISCA '24),* 1997.

[11] L.P. Deutsch and A.M. Schiffman, "Efficient Implementation of the Smalltalk-80 System," *Proc. Symp. Principles of Programming Languages (POPL '84),* 1984.

[12] K. Driesen and U. Hölzle, "Accurate Indirect Branch Prediction," *Proc. 25th Ann. Int'l Symp. Computer Architecture (ISCA '98),* 1998.

[13] K. Driesen and U. Hölzle, "The Cascaded Predictor: Economical and Adaptive Branch Target Prediction," *Proc. 31st Ann. ACM/IEEE Int'l Symp. Microarchitecture (MICRO '31),* 1998.

[14] K. Driesen and U. Hölzle, "Multi-Stage Cascaded Prediction," *Proc. European Conf. Parallel Processing,* 1999.

[15] M.A. Ertl and D. Gregg, "Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters," *Proc. Conf. Programming Language Design and Implementation (PLDI '03),* 2003.

[16] 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," *Proc. 25th Ann. Int'l Symp. Computer Architecture (ISCA '25),* 1998.

[17] The GAP Group, *GAP System for Computational Discrete Algebra,* http://www.gap-system.org/, 2007.

[18] C. Garrett, J. Dean, D. Grove, and C. Chambers, "Measurement and Application of Dynamic Receiver Class Distributions," Technical Report UW-CS 94-03-05, Univ. of Washington, Mar. 1994.

[19] GCC-4.0. GNU Compiler Collection, http://gcc.gnu.org/, 2007.

[20] 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 J.,* vol. 7, no. 2, May 2003.

[21] D. Grove, J. Dean, C. Garrett, and C. Chambers, "Profile-Guided Receiver Class Prediction," *Proc. Tenth Ann. Conf. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '95),* 1995.

[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 J.,* Feb. 2001, Q1 2001 Issue.

[23] U. Hölzle, C. Chambers, and D. Ungar, "Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches," *Proc. European Conf. Object-Oriented Programming (ECOOP '91),* 1991.

[24] U. Hölzle and D. Ungar, "Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '94),* 1994.

[25] Intel Corporation, ICC 9.1 for Linux, http://www.intel.com/cd/software/products/asmo-na/eng/compilers/284264.htm, 2007.

[26] Intel Corporation, Intel Core Duo Processor T2500, http://processorfinder.intel.com/Details.aspx?sSpec=SL8VT, 2007.

[27] Intel Corporation, *Intel VTune Performance Analyzers,* http://www.intel.com/vtune/, 2007.

[28] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani, "A Study of Devirtualization Techniques for a Java Just In-Time Compiler," *Proc. 15th ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00),* 2000.

[29] D.A. Jiménez and C. Lin, "Dynamic Branch Prediction with Perceptrons," *Proc. Seventh Int'l Symp. High Performance Computer Architecture (HPCA '00),* 2001.

[30] J.A. Joao, O. Mutlu, H. Kim, R. Agarwal, and Y.N. Patt, "Improving the Performance of Object-Oriented Languages with Dynamic Predication of Indirect Jumps," *Proc. 13th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '08),* 2008.

[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," *Proc. 18th Ann. Int'l Symp. Computer Architecture (ISCA '91),* 1991.

[33] J. Kalamatianos and D.R. Kaeli, "Predicting Indirect Branches via Data Compression," *Proc. 31st Ann. ACM/IEEE Int'l Symp. Microarchitecture (MICRO '98),* 1998.

[34] R.E. Kessler, "The Alpha 21264 Microprocessor," *IEEE Micro,* vol. 19, no. 2, pp. 24-36, Mar./Apr. 1999.

[35] H. Kim, J.A. Joao, O. Mutlu, C.J. Lee, Y.N. Patt, and R. Cohn, "VPC Prediction: Reducing the Cost of Indirect Branches via Hardware-Based Dynamic Devirtualization," *Proc. 34th Ann. Int'l Symp. Computer Architecture (ISCA '07),* 2007.

[36] 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," *Proc. 39th Ann. IEEE/ACM Int'l Symp. Microarchitecture (MICRO '06),* 2006.

[37] J.K.F. Lee and A.J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *Computer,* vol. 17, no. 1, Jan. 1984.

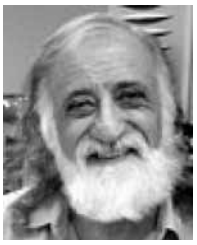
[38] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," *Proc. Programming Language Design and Implementation (PLDI '05),* 2005.
[39] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A Full System Simulation Platform," *Computer,* vol. 35, no. 2, pp. 50-58, Feb. 2002.
[40] T. McDonald, "Microprocessor with Branch Target Address Cache Update Queue," US patent 7,165,168, 2007.
[41] S. McFarling, "Combining Branch Predictors," Technical Report TN-36, Digital Western Research Laboratory, June 1993.
[42] Microsoft Research, Bartok Compiler, http://research.microsoft.com/act/, 2007.
[43] V. Morrison, "Digging into Interface Calls in the .NET Framework: Stub-Based Dispatch," http://blogs.msdn.com/vancem/archive/2006/03/13/550529.aspx, 2007.
[44] D. Novillo, Personal communication, Mar. 2007.
[45] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation," *Proc. 37th Ann. IEEE/ACM Int'l Symp. Microarchitecture (MICRO '04),* 2004.
[46] A. Roth, A. Moshovos, and G.S. Sohi, "Improving Virtual Function Call Target Prediction via Dependence-Based Pre-Computation," *Proc. Int'l Conf. Supercomputing (ICS '99),* 1999.
[47] A. Seznec and P. Michaud, "A Case for (Partially) Tagged Geometric History Length Branch Prediction," *J. Instruction-Level Parallelism (JILP),* vol. 8, Feb. 2006.
[48] D. Tarditi, Personal communication, Nov. 2006.
[49] J. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy, "POWER4 System Microarchitecture," *IBM Technical White Paper,* Oct. 2001.
[50] M. Wolczko, *Benchmarking Java with the Richards Benchmark,* http://research.sun.com/people/mario/java_benchmarking/richards/richards.html, 2007.
[51] T.-Y. Yeh, D. Marr, and Y.N. Patt, "Increasing the Instruction Fetch Rate via Multiple Branch Prediction and Branch Address Cache," *Proc. Seventh Int'l Conf. Supercomputing (ICS '93),* 1993.

Hyesoon Kim received the BA degree in mechanical engineering from Korea Advanced Institute of Science and Technology (KAIST), the MS degree in mechanical engineering from Seoul National University, and the MS and PhD degrees in computer engineering from the University of Texas at Austin. She is an assistant professor in the School of Computer Science at Georgia Institute of Technology. Her research interests include high-performance energy-efficient heterogeneous architectures, and programmer-compiler-microarchitecture interaction. She is a member of the IEEE and the IEEE Computer Society.

José A. Joao received the Electronics Engineer degree from the Universidad Nacional de la Patagonia San Juan Bosco, Argentina, and the MS degree in ECE from the University of Texas at Austin, where he is currently working toward the PhD degree in computer engineering. He is interested in computer architecture, with a focus on high-performance energy-efficient microarchitectures, compiler-microarchitecture interaction, and architectural support for programming languages. He is an assistant professor (on leave) at the Universidad Nacional de la Patagonia San Juan Bosco. He worked at Microsoft Research during summers 2007-2008 and at AMD during summer 2005. He was a recipient of the Microelectronics and Computer Development Fellowship from the University of Texas at Austin during 2003-2005. He is a student member of the IEEE and the IEEE Computer Society.

Onur Mutlu received the BS degrees in computer engineering and psychology from the University of Michigan, Ann Arbor, and the MS and PhD degrees in ECE from the University of Texas at Austin. He is an assistant professor of ECE at Carnegie Mellon University. He is interested in computer architecture and systems research, especially in the interactions between languages, operating systems, compilers, and microarchitecture. Prior to Carnegie Mellon, he worked at Microsoft Research, Intel Corporation, and Advanced Micro Devices. He was a recipient of the Intel PhD Fellowship in 2004, the University of Texas George H. Mitchell Award for Excellence in Graduate Research in 2005, and the Microsoft Gold Star Award in 2008. He is a member of the IEEE and the IEEE Computer Society.

Chang Joo Lee received the BS degree in electrical engineering from Seoul National University in 2001, and the MS degree in computer engineering in 2004 from the University of Texas at Austin, where he is currently working toward the PhD degree in computer engineering. He is interested in computer architecture research mainly focused on high-performance memory systems and energy-efficient microarchitectures. He was a recipient of the scholarship from the Ministry of Information and Communication in Korea during 2002-2006 and the IBM PhD Fellowship in 2007. He is a student member of the IEEE.



Yale N. Patt received the BS degree from Northeastern, and the MS and PhD degrees from Stanford, all in electrical engineering. He is the Ernest Cockrell, Jr., Centennial chair in engineering and a professor of ECE at Texas. He continues to thrive on teaching the large (400 students) freshman introductory course in computing and advanced graduate courses in microarchitecture, directing the research of eight PhD students, and consulting in the microprocessor industry. He is the coauthor of *Introduction to Computing Systems*: *From Bits and Gates to C and Beyond* (McGraw-Hill, second edition, 2004), his preferred approach to introducing freshmen to computing. He has received a number of awards for his research and teaching, including the IEEE/ACM Eckert-Mauchly Award for his research in microarchitecture and the ACM Karl V. Karlstrom Award for his contributions to education. More details can be found on his Web site http://www.ece.utexas.edu/patt. He is a fellow of the IEEE and the IEEE Computer Society.

Robert Cohn received the PhD degree from Carnegie Mellon in 1992. He is a senior principal engineer at Intel where he works on binary translation and instrumentation. He developed Pin, a popular tool for dynamic instrumentation. Previously, he was a developer of Spike, a postlink optimizer for the Itanium and Alpha processors. At Digital Equipment Corporation, he implemented profile-guided optimization in the Alpha C compiler.