Adapting Branch-Target Buffer to Improve the Target Predictability of Java Code

TAO LI

University of Florida

RAVI BHARGAVA

Advanced Micro Devices

and

LIZY KURIAN JOHN

University of Texas at Austin

Java programs are increasing in popularity and prevalence on numerous platforms, including high-performance general-purpose processors. The success of Java technology largely depends on the efficiency in executing the portable Java bytecodes. However, the dynamic characteristics of the Java runtime system present unique performance challenges for several aspects of microarchitecture design. In this work, we focus on the effects of indirect branches on branch-target address prediction performance. Runtime bytecode translation, just-in-time (JIT) compilation, frequent calls to the native interface libraries, and dependence on virtual methods increase the frequency of polymorphic indirect branches. Therefore, accurate target address prediction for indirect branches is very important for Java code.

This paper characterizes the indirect branch behavior in Java processing and proposes an adaptive branch-target buffer (BTB) design to enhance the predictability of the targets. Our characterization shows that a traditional BTB will frequently mispredict a few polymorphic indirect branches, significantly deteriorating predictor accuracy in Java processing. Therefore, we propose a rehashable branch-target buffer (R-BTB), which dynamically identifies polymorphic indirect branches and adapts branch-target storage to accommodate multiple targets for a branch.

The R-BTB improves the target predictability of indirect branches without sacrificing overall target prediction accuracy. Simulations show that the R-BTB eliminates 61% of the indirect branch mispredictions suffered with a traditional BTB for Java programs running in interpreter mode (46% in JIT mode), which leads to a 57% decrease in overall target address misprediction rate (29% in JIT mode). With an equivalent number of entries, the R-BTB also outperforms the previously proposed

This research was supported in part by the National Science Foundation under grant numbers 0113105 and 9807112, and by Tivoli, Motorola, Intel, IBM, and Microsoft Corporation.

Authors' addresses: Tao Li, Department of Electrical and Computer Engineering, University of Florida, Gainesville, Florida, 32611; email: taoli@ece.ufl.edu; Ravi Bhargava, Advanced Micro Devices, Austin, Texas, 78741; email: ravi.bhargava@amd.com; Lizy Kurian John, Department of Electrical and Computer Engineering, University of Texas at Austin, Austin, TX 78712; email: ljoh@ece.utexas.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.

© 2005 ACM 1544-3566/05/0600-0109 \$5.00

target cache scheme for a majority of Java programs by adapting to a greater variety of indirect branch behaviors.

Categories and Subject Descriptors: C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Adaptable architectures*; D.3 [**Programming Languages**]: Processors—*Run-time environments*

General Terms: Design, Performance

Additional Key Words and Phrases: Computer architecture, Java, pipelined processor, branch prediction, branch-target buffer (BTB)

1. INTRODUCTION

With the “write-once, run-anywhere” philosophy, Java applications are now prevalent on numerous platforms. This popularity has led to an increase in Java processing on general-purpose processors as well. The Java runtime system is the cornerstone of Java technology [Lindholm and Yellin 1999]. However, this system has unique execution characteristics that pose new challenges for high-performance design. One such area is branch-target prediction. While many branches are easily predicted, indirect branches that jump to multiple targets are among the most difficult branches to predict with conventional branch target-prediction hardware.

The current generation of microprocessors ubiquitously supports speculative execution by predicting the outcomes of the control-flow transfer in programs [Yeh and Patt 1995]. The trend toward wide-issue and deeply pipelined designs increases the penalty for mispredicting control transfer. For example, a mispredicted branch costs about 10 cycles on the Pentium III and Athlon and 20 cycles on the Pentium 4. Therefore, accurate control-flow prediction is a critical performance issue on current and future microprocessors. Current processors (e.g., Pentiums, Athlon, Alpha 21264, and Itanium 2) predict branch targets with a branch-target buffer (BTB) or similar structure, which caches the most recently resolved target [Lee and Smith 1984]. Most indirect branches are unconditional jumps and predicting their branch direction is trivial. Therefore, indirect branch-prediction performance is largely dependent on the target address prediction accuracy.

The execution of Java programs and the Java Runtime Environment results in more frequent indirect branching compared to other commonly studied applications [Li and John 2001; Li et al. 2002]. Runtime interpretation and just-in-time (JIT) compilation of bytecodes performed by the Java Virtual Machine (JVM) are subject to high indirect branch frequency. Common sources are switch statements and the numerous indirect function calls [Driesen and Hölzle 1998a]. Moreover, to facilitate the modularity, flexibility, and portability paradigms, many Java native interface routines are coded as dynamically shared libraries. Calls to these routines are implemented as indirect function calls. Finally, as an object-oriented programming language, Java implements virtual methods to promote code reuse. A virtual method is a function that allows polymorphic implementations for it. Virtual subroutines execute indirect branches using virtual method tables, that create additional indirect branches for most Java compilers.

Table I. Indirect Branch Frequency in Java and C Programs^a

Java (SPEC JVM98)	Interpretation		Just-in-Time Compilation	
	% in Total Instructions	% in Branch Instructions	% in Total Instructions	% in Branch Instructions
<i>db</i>	3.0	19.4	2.5	15.1
<i>jess</i>	3.3	21.4	2.4	9.7
<i>javac</i>	2.6	15.7	1.9	13.3
<i>jack</i>	2.5	15.1	2.1	11.4
<i>mtrt</i>	2.7	19.2	2.0	17.3
<i>compress</i>	4.3	27.0	1.6	7.8
<i>AVG</i>	3.1	19.6	2.1	12.4
(SPEC INT95 C)	% in Total Instructions	% in Branch Instructions		
<i>go</i>	0.7	6.0		
<i>compress</i>	0.4	3.0		
<i>m8ksim</i>	0.8	7.0		
<i>gcc</i>	1.1	6.1		
<i>jpeg</i>	0.2	3.1		
<i>li</i>	2.0	14.3		
<i>perl</i>	2.2	13.9		
<i>vortex</i>	0.9	8.0		
<i>AVG</i>	1.2	8.7		

^aThe indirect branch instruction mix ratios in Java programs are presented for runs in interpreter-only and JIT modes.

Previous studies have concentrated mainly on the analysis and optimization of indirect branch prediction for SPEC integer and C++ programs [Calder et al. 1994; Chang et al. 1997; Driesen and Hölzle 1998a, 1998b]. Driesen and Calder [Driesen and Hölzle 1998a; Calder et al. 1994a] studied the indirect branch frequency using a suite of large C++ applications. It was observed that those C++ programs execute an indirect branch as frequently as once every 50 instructions. Driesen et al. also predicted that Java programs (where all nonstatic calls are virtual) are likely to use indirect calls even more frequently.

Table I compares the indirect branch frequency found in Java processing with that found in the SPEC INT95 C benchmarks. The indirect branch frequencies for Java are uniformly high, while only C programs that perform code compilation or interpretation (*gcc*, *li*, and *perl*) show high indirect branch frequency. Previous studies [Driesen and Hölzle 1998b] have shown similar results. On average, 20% of branches in Java (in interpreter mode) are indirect branches, while only 9% are indirect branches in the SPEC INT95 C benchmarks. Compared with the C++ indirect branch frequency reported by Driesen and Calder, the Java workloads studied here execute indirect branches more frequently.

Employing a complete system simulation framework, we further characterize the indirect branches in Java and study their impact on the underlying branch prediction hardware. Our characterization shows that a few critical polymorphic indirect branches can significantly deteriorate the BTB performance during Java execution. For example, 10 indirect branch sites are responsible for 75% of indirect branch mispredictions during Java code execution. Therefore, a solution that can effectively handle target prediction for a small number of polymorphic branch sites could improve the BTB performance.

We propose a rehashable BTB (R-BTB) scheme, which identifies critical polymorphic indirect branches and remembers them in a small separate structure called the critical indirect branch instruction buffer (CIBIB). Targets for polymorphic branches promoted to the CIBIB are found by rehashing into the R-BTB target storage. This novel rehashing function allows polymorphic branch targets to use the same resources as monomorphic branches without reducing overall branch-target prediction accuracy. Simulations using SPEC JVM98 reveal that the R-BTB eliminates a significant portion of the indirect branch mispredictions versus a traditional BTB while reducing the overall branch-target misprediction rate in both interpreter and JIT modes. In addition, the R-BTB outperforms an indirect branch-target cache and BTB combination (target cache [Chang et al. 1997]) with comparable resources for five of the six Java benchmarks studied.

The rest of this paper is organized as follows. Section 2 describes the simulation-based experimental setup and the Java benchmarks. Section 3 provides insight into the indirect branch characteristics of Java execution. Section 4 presents the R-BTB design. Section 5 explores the R-BTB design trade-offs. Section 6 evaluates the performance of the R-BTB by comparing its misprediction rate with that of a traditional BTB scheme and a combined BTB/target cache scheme. Section 7 discusses the related work. Finally, Section 8 summarizes the conclusions of this paper.

2. EXPERIMENTAL METHODOLOGY AND BENCHMARKS

This section describes the simulation-based experimental setup and Java benchmarks used to evaluate the proposed R-BTB scheme. To analyze the entire execution of the JVM and Java workloads, we use the SimOS full-system simulation framework [Rosenblum et al. 1995] to study Java indirect branch characteristics. The simulation environment uses the IRIX 5.3 operating system. The Sun Java Development Kit ported by Silicon Graphics Inc. provides the Java runtime environment. The SPEC JVM98 [SPEC JVM98] suite described in Table II is used for this research.¹

We collect the system traces from a heavily instrumented SimOS MXS simulator and then feed them to our back-end simulators and profiling tool sets, which have been used for several of our research studies [Li et al. 2000; Li and John 2001]. We simulate each benchmark (with the S1 dataset) on the SimOS MXS model until completion, except for the benchmark compress running in interpreter-only mode. In this case, we use the first 2000M instructions. Table II reports the number of static and dynamic indirect branch call sites collected from our complete system simulation. Call returns are excluded because they can be predicted accurately with a return address stack. The execution of these benchmarks in both the JIT compiler (jit) and interpreter-only (intr) modes is analyzed. Choosing between the JIT and interpreter modes requires complex space and performance tradeoffs. Interpretation is still commonly used

¹We exclude the benchmark mpegaudio from our experiments because it failed to execute on the detailed model of SimOS.

Table II. SPEC JVM98 Benchmarks and Their Indirect Branch Statistics

Benchmarks	Description		Indirect Branch Statistics	
			Static Sites	Dynamic Instances
<i>db</i>	Performs multiple database functions on a memory resident database	<i>jit</i>	5,786	2,514,766
		<i>intr</i>	4,116	2,815,831
<i>jess</i>	Java expert shell system based on NASA's CLIPS expert system	<i>jit</i>	7,249	7,496,669
		<i>intr</i>	4,205	11,132,086
<i>javac</i>	The JDK 1.0.2 Java compiler compiling 225,000 lines of code	<i>jit</i>	7,219	5,305,566
		<i>intr</i>	4,266	5,176,207
<i>jack</i>	Parser generator with lexical analysis, early version of what is now JavaCC	<i>jit</i>	7,480	31,348,141
		<i>intr</i>	3,998	31,037,413
<i>mtrt</i>	Dual-threaded raytracer	<i>jit</i>	7,015	24,523,313
		<i>intr</i>	4,097	54,533,384
<i>compress</i>	Modified Lempel-Ziv method (LZW) to compress and decompress large file	<i>jit</i>	5,726	36,698,515
		<i>intr</i>	3,964	40,948,296

in state-of-the-art Java technologies and on resource-constrained platforms [Radhakrishnan et al. 2000], so we present analysis for both scenarios.

Note that the data in Table II are the statistics of the active indirect branches recorded during program execution. The JIT compiler has a larger instruction footprint than the interpreter. Therefore, it shows larger active indirect branch sites. Table II also shows that the JIT mode can execute more dynamic instances of indirect branches than the interpreter mode. This is because in the JIT mode, the dynamic instances of indirect branch are largely determined by the fraction of JVM bytecode translation versus the native code execution. This fraction varies on different benchmarks.

3. CHARACTERIZATION OF INDIRECT BRANCHES IN JAVA

In this section, we present our characterization of indirect branches in Java. The following analysis is performed with the JVM running in both the interpreter and JIT modes.

Current processors predict branch targets with a BTB, which caches the most recently resolved target. An indirect branch can always be predicted “taken” by setting a corresponding “branch-type” bit once it enters the BTB. Indirect branch-prediction performance is largely dependent on the efficiency of the BTB, because the BTB is referred for obtaining the target address at the fetch cycle of a pipeline.

Will the conventional BTB structure work well with the indirect branch-rich Java runtime system? To answer this question, we perform a BTB miss-rate study (4-way set associative, with table size varied from 256- to 8K-entry). The BTB performance is examined by executing each benchmark with both JIT compiler and interpreter. The results are plotted in Figure 1. The BTB miss rate shown in Figure 1 is further separated into tag miss caused by a BTB entry absence and target miss caused by incorrect targets in the BTB entry.

As shown in Figure 1, The BTB performance is largely dependent on the JVM style as interpretation significantly increases the BTB miss rate on most studied benchmarks. For example, the BTB miss rate in benchmark compress

Fig. 1. BTB performance on Java processing.

Fig. 2. Dynamic indirect branch-target distribution.

increases from 8% in JIT mode to 96% in interpreting mode. Previous study [Li and John 2001] shows when *compress* is executed with the interpreter, branch instances repeatedly work through a body of distinct address sites. The branch-target transfer patterns almost always cause misprediction in a BTB where only the most recently transferred target is recorded. Figure 1 further shows that the Tag miss rate becomes negligible as the BTB size increases. However, increasing BTB size does not reduce the BTB target miss rate significantly.

3.1 Polymorphic versus Monomorphic Indirect Branches

Indirect branches can be categorized as branches that only jump to one target during the course of execution (monomorphic branches) and those that jump to multiple targets (polymorphic branches). Polymorphic branches are the ones that make indirect branch-target prediction difficult. Figure 2 reports the percentage of monomorphic ($\text{target} = 1$) and polymorphic ($\text{targets} \geq 2$) indirect branch execution. The remaining bars illustrate the degree of polymorphism. In the interpreter mode, over 50% of the executed indirect branches are, on average, polymorphic. This is primarily due to a switch statement in the bytecode translation routine of the interpreter. In JIT mode, JVM spends no time interpreting bytecodes, but 25% of indirect branch execution is still polymorphic.

Figure 2 shows that in Java processing, polymorphic indirect branch execution can be significant. Figure 3 shows that the number of static polymorphic branches is quite small, i.e., less than 5% of all indirect branch sites. Therefore, a small buffer can capture many of the polymorphic indirect branches. This observation is exploited later in Section 4 when designing the R-BTB is discussed.

Fig. 3. Static indirect branch site target distribution.

Fig. 4. Impact of critical polymorphic indirect branches. Benchmarks run in interpreter mode. BTB is 4-way with 2K-entry.

3.2 Impact of Polymorphic Indirect Branches

A few polymorphic indirect branches can impact target prediction performance significantly. In this paper, we use the term “critical indirect branches” to refer those static branches whose execution has the most significant impact on the overall branch-prediction accuracy. Figure 4 shows the misprediction rate for indirect branches using a traditional BTB. The top portion of each bar represents the fraction of mispredictions due to the 10 most critical indirect branches.

Table III summarizes the most critical polymorphic indirect branch identified in the studied JVM (with the interpretation execution mode). These polymorphic indirect branches come from code-performing bytecode interpretation (`<ExecuteJava>`), calls to the dynamically shared native interface libraries (`<sgi_to_tran_w>`, `<sgi_invokeinterface>`), and other JVM management routines (`<monitorExit>`, `<createPrimitiveClass>` and `<FreeClass>`). These critical polymorphic indirect branches show highly interleaved target-transfer patterns, which can not be predicted accurately with a conventional BTB structure [Li and John 2001].

4. REHASHABLE BTB (R-BTB)

The previous section reveals that polymorphic indirect branches lead to a high misprediction rate on a conventional BTB structure. Simply tracking the most recently used target is not sufficient to capture multiple target addresses. In this section, we propose a BTB enhancement to improve the target predictability of polymorphic branches. We begin by supplying a brief overview of the target cache, an existing scheme aimed at improving the target predictability of indirect branches [Chang et al. 1997].

Table III. Most Frequently Invoked Polymorphic Indirect Branches in JVM Interpretation

<Routine+Offset>	% of Polymorphic Indirect Branch Instances					
	<i>db</i>	<i>jess</i>	<i>javac</i>	<i>jack</i>	<i>mtrt</i>	<i>compress</i>
<ExecuteJava+b8>	63.09	82.66	53.00	75.62	87.66	97.14
<ExecuteJava+357c>	6.10	6.38	6.35	8.11	11.06	2.00
<monitorExit+34>	5.63	—	9.12	—	—	—
<createPrimitiveClass+840>	3.28	1.09	4.20	—	—	—
<sgi_to_tran_w+58>	2.57	1.68	3.53	5.25	—	—
<sgi_to_tran_w+2c>	2.35	1.62	3.69	4.92	—	—
<FreeClass+3a0>	2.21	—	2.55	—	—	—
<malloc+38>	1.87	—	2.01	—	—	—
<Str2ID+6c>	1.30	—	1.34	—	—	—
<free+38>	1.25	—	1.47	—	—	—
<invokeNativeMethod+c8>	1.23	—	—	—	—	—
<sgi_invokeinterface+58>	—	—	3.25	—	—	—
<sgi_invokevirtual_object+24>	—	—	1.32	1.33	—	—
Subtotal (% of polymorphic indirect branch execution)	90.88	93.43	91.83	95.23	98.72	99.14

Fig. 5. Target cache (TC) scheme.

4.1 Target Cache

The target-cache scheme (shown in Figure 5) attempts to distinguish different dynamic occurrences of each indirect branch by exploiting the branch-target history of indirect branches. The assumption is that the target of a polymorphic indirect branch depends on the global program path taken prior to the branch. The BTB and target cache are accessed simultaneously. If an indirect branch is identified, the target address is taken from the target cache. Otherwise, the BTB produces the branch target.

In the target-cache scheme, the number of entries allocated to the BTB and to the target cache is determined at design time. Because the indirect branch frequency changes between different programs, it is possible that the target-cache

Fig. 6. Rehashable BTB (R-BTB).

resources are not always utilized efficiently. Our characterization of indirect branches in Java suggests that while the number of dynamic polymorphic targets varies widely between programs, the static number of polymorphic indirect branch sites is consistently low.

4.2 Rehashable BTB Design

We propose a R-BTB (shown in Figure 6), which employs a small structure, the critical indirect branch instruction buffer (CIBIB), to identify the performance-critical polymorphic indirect branches. A more detailed analysis of CIBIB structure and associativity is discussed in Section 5. Once these critical branches are identified, their targets are rehashed into multiple, separate entries in the R-BTB. Like the target cache, the R-BTB uses a target-history register (THR) to collect path history. The path history in the THR is hashed with the critical-branch PC to identify an entry in the R-BTB. The primary difference between the R-BTB and the target-cache mechanism is that instead of using separate structures for storing the targets of indirect branches and the targets of direct branches, the R-BTB uses the same structure. Therefore, the resources allocated to target prediction can be shared dynamically based on the frequency of polymorphic indirect branches instead of split statically based on a predetermined configuration.

As depicted in Figure 6, a CIBIB entry consists of a field for identifying critical branches. The branch-target storage is similar to a traditional BTB augmented with a target-miss counter (TMC). The TMC is incremented if a branch that hits in the BTB receives an incorrect target prediction. Once the TMC reaches a certain threshold, the branch is promoted to the CIBIB and its entry in the target storage is reclaimed.

Branches that reside in the CIBIB are critical polymorphic indirect branches. The R-BTB is still used to store their targets, but not in the traditional manner. Instead, the THR value is XORed with bits from the branch PC to choose a R-BTB entry. The target-history register stores a concatenation of partial

target addresses. The THR can be maintained globally or locally. In a global configuration (as illustrated in Figure 6), the THR is updated with the targets of branches contained in the CIBIB, and all critical polymorphic branch sites share the same THR. In a local configuration, separate target patterns are maintained for each polymorphic branch site residing in the CIBIB. We investigate target prediction accuracies of using different THR structures in Section 5.

4.3 Target Prediction with the R-BTB

This section provides a detailed example of target prediction using the R-BTB. Figure 6 is a corresponding illustration of this process. When a branch target is being predicted, the PC of the branch is sent to the CIBIB. If it hits in the CIBIB, the path-history pattern collected in the THR, along with the PC, is used to generate the address for R-BTB access. If the branch misses in the CIBIB, its PC is used as the address to access the R-BTB.

At runtime, the PCs of critical indirect branches with high BTB misprediction rate are dynamically filtered out to the CIBIB. Whenever a branch instruction hits a BTB entry, but the predicted target is incorrect, the TMC is increased [as shown in Figure 7(b)]. When a TMC reaches the given threshold, a filtering occurs [Figure 7(c)]. The BTB entry is then reclaimed for future use and the corresponding TMC is reset [Figure 7(d)]. The indirect branch's target information is stored into the rehashed entry of the BTB [Figure 7(e)]. When the actual branch target is resolved in pipeline, the target-history register is updated (if necessary) and the new resolved target bits are coalesced to keep tracking-history pattern information. By using the more precise target-history pattern as a rehashing function, the multiple targets of the critical polymorphic indirect branches can be stored into and predicted with different BTB entries [Figure 7(f)]. In this manner, the R-BTB houses targets of both indirect and direct branches.

5. R-BTB DESIGN TRADEOFFS

As the extension to a BTB scheme, one of the design issues is to keep the overhead of R-BTB small enough by avoiding significant hardware budget increase. In this section, we explore the design trade-offs of the R-BTB to search for cost-effective configurations.

5.1 Threshold of the TMC Counter

The proposed R-BTB uses CIBIB to store and identify the critical polymorphic indirect branches. If the capacity of CIBIB is too small, there will be a significant thrashing effect between different critical polymorphic indirect branch sites. On the other hand, CIBIB can be underutilized and will cause significant access latency if its size is too large. Inherently, the number of CIBIB entries is decided by the number of unique branch sites that can be promoted on a given threshold of the TMC counters (Th_{TMC}). To investigate this design tradeoff, we simulate a R-BTB with an infinite-size CIBIB to find the maximum number of branch sites that can be identified for a given Th_{TMC} (data is shown in Table IV).

Fig. 7. Target prediction with R-BTB.

Table IV shows that the number of unique branch sites that can potentially be promoted ranges from 13 (in compress with $Th_{TMC} = 512$) to 74 (in javac with $Th_{TMC} = 16$). Increasing Th_{TMC} decreases the promotion capability for polymorphic branches. Notice that very small TMCs may require too large a CIBIB to accommodate the numerous promoted branches.

5.2 Global THR versus Local THR

We examine the efficiency of using local versus global target-pattern history, as described below. To reduce the impact of capacity miss on performance, we use 128-entry CIBIB with four-way set associativity in our simulation. (We

Table IV. Number of Promoted Branch Sites

Benchmarks	Th _{TMC} = 16	Th _{TMC} = 32	Th _{TMC} = 64	Th _{TMC} = 128	Th _{TMC} = 256	Th _{TMC} = 512
<i>db</i>	64	54	45	41	35	29
<i>jess</i>	71	61	54	45	36	30
<i>javac</i>	74	61	50	38	29	24
<i>jack</i>	66	54	47	42	36	34
<i>mtrt</i>	65	49	37	27	21	21
<i>compress</i>	52	34	24	19	15	13
AVG	65	52	43	35	29	25

Fig. 8. Global THR vs. local THR.

later show the performance of the proposed scheme with resource-constrained configurations.) In addition, a $\text{Th}_{\text{TMC}} = 512$ is set for the evaluation. We model a R-BTB using global THR and a R-BTB using local THR. Both R-BTBs use XOR as the rehashing function. The information stored in the THR entry is used to access the four-way set-associative BTB. The misprediction rate for the indirect branch only and for all branch instructions (in interpretation mode) are given in Figure 8, which shows that the use of local history-based THR does not necessarily imply a better performance compared with its global history THR counterpart. For this reason, we use global THR configuration for our further design space explorations.

To further search for the cost-effective R-BTB design, we set up experiments to investigate the performance of resource-constrained configurations. Figures 9(a) and 9(b) reveal the performance of the R-BTB with CIBIB entries varied from 16 to 128 and associativity from 1 to 4 (on a 32-entry CIBIB). We find that increasing CIBIB entries does not provide significant prediction accuracy improvement (less than 2%) and the use of a 16-entry, direct-mapped CIBIB can provide performance improvement comparable to that of more complex and costly configurations.

Figure 10 further shows the impact of TMC thresholds on the R-BTB misprediction rate (with a 32-entry, DM CIBIB). Increasing the promotion threshold is found to slightly reduce misprediction rates. It is observed that most of the highly invoked polymorphic indirect branches with high target-miss rate can be easily captured by a promotion threshold of 32.

Fig. 9. Impact of CIBIB size and set associativity on the indirect branch prediction accuracy.

Fig. 10. Impact of TMC threshold.

6. PERFORMANCE EVALUATION OF THE R-BTB

In this section, we present the performance of a traditional BTB, a target-cache scheme, and the R-BTB. The indirect branch misprediction rate and the overall branch prediction rate are compared for the different target prediction mechanisms. To illustrate the benefits of dynamic target storage allocation, several static configurations of the target-cache scheme are also analyzed.

6.1 Evaluated Target Predictors

In Section 5, we examined the impact of several R-BTB factors, such as THR entry configuration, TMC threshold, CIBIB size, and CIBIB associativity. Based on our experiments, we use a global THR, a TMC with a threshold of 512, and a 16-entry, direct-mapped CIBIB for our performance evaluation. The least significant three bits of the target address (bits 2–4, since bits 0–1 are always zero) are recorded and concatenated in the THR. The simple and small CIBIB configuration is chosen to reduce access latency. The other design parameters are optimized for performance.

All branch-target prediction structures are allocated about 2048 entries [Chang et al. 1997; Driesen and Hölzle 1998a] and are four-way set associative, unless specified. The target-cache scheme shares resources evenly between a BTB and the target cache. We found that this is the best performing combination for the Java benchmarks, as discussed further in Section 6.4.

The target predictors in this section are also used to predict branch targets for branch types other than indirect branches. In addition to indirect

Table V. Indirect Branch-Target Misprediction Rates (Interpreter Mode)

Structures	Size (# of entries)	Indirect Misprediction Rate (%)					
		<i>db</i>	<i>jess</i>	<i>javac</i>	<i>jack</i>	<i>mtrt</i>	<i>compress</i>
BTB	2048	21.1	43.8	18.5	41.8	65.7	96.3
BTB + Tagged TC	1024 + 1024	11.0	20.4	15.0	28.7	21.7	13.2
R-BTB	2048 + 16-entry, DM CIBIB	7.9	20.1	8.6	23.2	21.0	29.7

Table VI. Indirect Branch-Target Misprediction Rates (JIT Compilation Mode)

Structures	Size (# of entries)	Indirect Misprediction Rate (%)					
		<i>db</i>	<i>jess</i>	<i>javac</i>	<i>jack</i>	<i>mtrt</i>	<i>compress</i>
BTB	2048	7.9	11.7	12.3	14.3	13.1	8.8
BTB + Tagged TC	1024 + 1024	10.8	13.7	12.5	13.8	6.7	1.1
R-BTB	2048 + 16-entry, DM CIBIB	4.6	6.7	6.1	10	3.8	5.6

Table VII. Overall Branch-Target Misprediction Rates (Interpreter Mode)

Structures	Size (# of entries)	Overall Branches Misprediction Rate (%)					
		<i>db</i>	<i>jess</i>	<i>javac</i>	<i>jack</i>	<i>mtrt</i>	<i>compress</i>
BTB	2048	3.2	9.8	3.4	10.6	14.8	35.2
BTB + Tagged TC	1024 + 1024	2.5	5.5	3.3	8.0	5.0	4.9
R-BTB	2048 + 16-entry, DM CIBIB	1.7	5.3	2.1	7.0	5.5	11.7

branches, taken conditional branches access the target predictors in our evaluation. In some architectures, target prediction is not always necessary for predicting fall-through paths of not-taken branches or for predicting direct branch targets.

6.2 Branch-Target Prediction Performance

Tables V and VI present the misprediction rates of indirect branch targets for the evaluated schemes in interpreter and JIT modes. The proposed R-BTB technique improves the misprediction rate for all of the benchmarks compared to a traditional BTB. On average, it reduces the misprediction rate versus a traditional BTB from 47.8 to 18.4% in interpreter mode and from 11.3 to 6.1% in JIT mode. The most drastic improvements are seen for the benchmarks *mtrt* and *compress*.

The R-BTB also improves the performance of indirect branches versus the target cache for five out of the six benchmarks. Only the program *compress* exhibits better target prediction for a target-cache scheme. While the average performance is the same in interpreter mode, the R-BTB improves the misprediction rate in JIT mode from 11.4 to 6.1%. In fact, in JIT mode the target cache does not always perform better than a traditional BTB.

Improving indirect branch-target prediction performance can only benefit the processor if the overall branch-target prediction performance is also improved. Tables VII and VIII present the overall branch-target misprediction rates for interpreter and JIT modes. Versus a traditional BTB, in both interpreter and JIT modes, the R-BTB improves overall branch performance for all of the benchmarks and, on average, reduces the overall branch-target

Table VIII. Overall Branch-Target Misprediction Rates (JIT Compilation Mode)

Structures	Size (# of entries)	Overall Branches Misprediction Rate (%)					
		<i>db</i>	<i>jess</i>	<i>javac</i>	<i>jack</i>	<i>mtrt</i>	<i>compress</i>
BTB	2048	1.5	2.8	2.4	3.1	2.4	2.8
BTB + Tagged TC	1024 + 1024	2.3	3.8	3.1	3.6	1.5	0.5
R-BTB	2048 + 16-entry, DM CIBIB	1.2	2.2	1.7	2.9	0.9	1.8

misprediction rate from 12.8 to 5.6% in interpreter mode and from 2.5 to 1.8% in JIT mode.

The R-BTB also outperforms the target-cache scheme. In interpreter mode, the R-BTB produces a better misprediction rate for four out of the six benchmarks. The target cache does much better for the benchmark *compress*, which leads to an average improvement over the R-BTB—a 4.8% misprediction rate for the target cache versus 5.5% for the R-BTB. However, the R-BTB outperforms the target-cache scheme for five out of six benchmarks in JIT mode and reduces the average overall branch-target misprediction rate from 2.4 to 1.8%. Once again, it is interesting to note that the target cache does worse than a traditional BTB for four of the six benchmarks. More comparisons on the target prediction accuracies of BTB, BTB + TC, and R-BTB can be found in Appendix.

6.3 Discussion of Performance Results

The performance results are different depending on the JVM mode of execution. In interpreter mode, 19.5% of all dynamic branches are indirect branches and 11.8% of all branches are polymorphic indirect branches. In this scenario, the target cache predicts indirect branch targets much better than a traditional BTB because it has dedicated half of its resources to handle indirect branches. Despite the reduction in resources for direct branches, the target-cache scheme still easily outperforms the BTB overall.

In JIT mode, 10.5% of dynamic branches are indirect branches and only 3.2% are polymorphic branches. In this case, the traditional BTB and target cache have about the same average performance, and the BTB actually does overall branch target prediction better for four of the six benchmarks. There are many fewer indirect branches than in interpreter mode, so the ability to predict direct branches is important. Therefore, the 2048 shared entries of the traditional BTB provide more benefit than the 1024 dedicated entries of the target cache.

The advantage of the R-BTB is that it adapts to both cases. It allocates 2048 entries of target storage for all types of branches like the traditional BTB. However, using the CIBIB, it is able to identify critical polymorphic branches and rehash the multiple targets in the common target storage. Therefore, when the number of indirect branches is low, the R-BTB behaves like a traditional BTB. When the number of polymorphic branches is high, the R-BTB then behaves in a similar manner to the target cache. On average, this adaptive behavior results in better overall target-prediction accuracy, as shown.

The benchmark *compress* is the exception. The target cache does the best job of predicting branch targets for *compress*. This program has one of the largest

Fig. 11. Impact of target cache resource partitioning. Performance is with respect to a 4-way, 2048-entry BTB. TC stands for target cache.

percentages of indirect branches: 3.8% of all dynamic branches in JIT mode and 26.25% in interpreter mode. The more important characteristic is that compress has the highest degree of polymorphism. While the target cache and R-BTB have similar hashing schemes (based on THR and PC), the R-BTB is sharing the indirect target storage with other branches which increases the chance for entry pollution or corruption. This is the scenario where smaller, dedicated indirect branch-target storage proves beneficial.

6.4 Dynamic Target Storage Allocation versus Static Target Allocation

Previous work with the target cache [Chang et al. 1997] splits resources differently than discussed in this paper. In previous work, the results are presented using a 2k-entry BTB with an additional target cache of 256, 512, and 1024 entries. However, these sizes are chosen based on the performance of C programs. As shown earlier, the SPEC CINT95 programs have a much lower percentage of indirect branches. In addition, the number of polymorphic branches and the degree of polymorphism are less for C programs versus Java programs. For example, the largest percentage of polymorphic branches (out of all branches) for a SPEC CINT95 program is 3.2% for *perl*, while the average for the JIT and interpreter modes of Java are 3.2 and 11.8%, respectively. The R-BTB is better equipped to handle this variation in indirect branch behavior from workload to workload.

Figure 11 further states the case for a dynamic and adaptive scheme. Four different resource partitions are presented for the combined BTB and target-cache scheme: 1024 + 1024 (as in Section 5.2), 2048 + 512 (as suggested in [Chang et al. 1997]), 2048 + 1024, and 2048 + 2048. In addition to the 4-way configuration used earlier, a 16-way associative target cache is presented. A 4096-entry R-BTB is also presented for comparison, in addition to the 2048-entry R-BTB, from the previous sections.

There are several important points to observe in this figure. The R-BTB does better than four-way target cache configurations with the same amount of target storage. In some cases, the R-BTB predicts branch targets more accurately

Table IX. Access Time^a

Structures	Size (# of entries)		Processing Technology (μm)	Access Time (ns)
BTB (4-way)	2048		0.18	2.3
			0.1	1.3
BTB (4-way) + tagless TC	1024 + 1024		0.18	2.2
			0.1	1.3
BTB (4-way) + tagged TC	1024 + 1024	4- way	0.18	2.7
			0.1	1.5
		16- way	0.18	3.1
			0.1	1.7
R-BTB	2048 + 16-entry, DM CIBIB		0.18	3.0
			0.1	1.6

^aResults based on Cacti 2.0 [Reinman et al. 2000] simulations.Table X. Energy Consumption per Access^a

Structures	Size (# of entries)		Processing Technology (μm)	Energy per Access (nj)
BTB (4-way)	2048		0.18	9.7
			0.1	3.8
BTB (4-way) + tagless TC	1024 + 1024		0.18	9.8
			0.1	4.1
BTB (4-way) + tagged TC	1024 + 1024	4- way	0.18	12.3
			0.1	5.4
		16- way	0.18	26.5
			0.1	12.5
R-BTB	2048 + 16-entry, DM CIBIB		0.18	10.6
			0.1	4.2

^aResults based on Cacti 2.0 simulations.

than a target cache with more target storage entries and/or more associativity. The target-cache configuration that is reported to do well on C programs (2048 + 512) does not do well for Java applications. This highlights the advantage of an R-BTB versus strategies that statically allocate target storage resources.

6.5 Access Latency and Power Consumption

As described in Section 4, the R-BTB uses additional hardware (e.g. CIBIB, TMC, and THR) to improve the target-prediction accuracy. The increment of hardware budget can affect the R-BTB access latency and power consumption. To evaluate the time and the energy cost of the R-BTB accesses, we use Cacti [Reinman and Jouppi 2000], an integrated cache-timing and power-estimation tool. We configure the Cacti tool to simulate the conventional BTB, the target caches, and the R-BTB structures. The R-BTB estimations include the overheads of CIBIB, TMC, and THR.

Tables IX and X list the access latency as well as the energy dissipation per access of BTB, BTB + TC, and R-BTB structures. As can be seen, both BTB + TC (tagged) and R-BTB increase the overall BTB access latency. Compared with the tagged TC, the use of 16-entry, direct-mapped CIBIB in the R-BTB design

does not increase its access latency significantly. For the configurations that the R-BTB shows better performance [e.g., R-BTB(2048) versus BTB(1024) + 16 Way TC(1024) in Figure 11], the R-BTB has less access latency. Table X shows that the energy dissipated per access of the R-BTB is found to be much less than BTB + TC configurations, which statistically allocate dedicated set-associative structures to store the targets of indirect branches.

7. RELATED WORK

Lee and Smith [1984] did an early indirect branch-prediction study, exclusively focusing on C code. As discussed earlier in this work, Chang et al. proposed several target cache schemes for indirect branches and their performance is evaluated using selected SPEC CINT95 programs. Calder and Grunwald [1994b] proposed the two-bit counter update rule for BTB target addresses and showed that it improved the prediction rate of a suite of C++ programs. Aigner and Dean [Aigner and Hölzle 1996; Dean et al. 1996] observed that in object-oriented programs, polymorphic branches occasionally switch their target, but are often dominated by one most frequent target. The BTB target prediction accuracy is quite poor on object-oriented programs.

Hsieh et al. [1997] studied the performance of Java code running in interpreter mode and observed that microarchitectural mechanisms, such as BTB, are not well utilized. However, their work does not provide an in-depth characterization on Java indirect branches. Ertl and Gregg [2001] investigated the performance of several virtual machine interpreters on several branch predictors and found that BTBs mispredict 81–98% of the indirect branches in switch-dispatch interpreters. A related study [Vijaykrishnan and Ranganathan 1999] examined the effectiveness of using path history to predict target addresses of indirect branches to counter the effects of virtual method invocations in Java. The results are presented for small Java programs (e.g., *richards* and *deltablue*) and do not apply directly to all JVM execution modes. Recently, Li and John [2001] characterized control-flow transfer in Java processing using a full-system simulation and SPEC JVM98 benchmarks. However, no hardware optimization was proposed.

There are a number of recent papers [Driesen and Hölzle 1998a; 1998b; Kalamatianos and Kaeil 2000; Vlaovic et al. 2000; Li et al. 2002] on improving indirect branch prediction. Driesen and Hölzle [1998a] investigated the performance of two-level and hybrid predictors dedicated exclusively to predicting indirect branch targets. Their work optimized for select SPEC CINT95 and C++ applications. However, like the target cache, this requires a static partitioning of target prediction resources. Driesen and Hölzle [1998b] also proposed a cascaded predictor, which dynamically classifies and filters polymorphic indirect branches from a simple first-stage BTB into a second-stage history-based buffer. The primary differences between the cascaded predictor and the R-BTB are: (1) the R-BTB has a stricter filtering criterion for determining important polymorphic branches (512 misses versus one), and (2) the R-BTB stores polymorphic branch targets in the same structure as the monomorphic branch targets. While Driesen and Hölzle suggest using both of these mechanisms

for indirect branches only, they could also be used for any type of target prediction.

Ertl and Gregg [2003] proposed software ways to improve the prediction accuracy of BTBs for interpreters: replicating virtual machine (VM) instructions and combining sequences of VM instructions into superinstructions.

8. CONCLUSION

Java execution results in more frequent execution of polymorphic indirect branches due to the nature of the language and the underlying runtime system. A traditional branch-target buffer (BTB) is not equipped to predict multiple targets for one static branch, while previous indirect target prediction work targets indirect branch prediction in C and C++ workloads. To achieve high branch-target prediction accuracy in Java execution, we propose a new rehashable BTB (R-BTB). Instead of statically allocating dedicated resources for indirect branches, the R-BTB dynamically identifies critical polymorphic indirect branches and rehashes their targets into unified branch-target storage. This method of dealing with polymorphic branches greatly reduces the number of indirect branch-target mispredictions as well as the overall target misprediction.

This paper first characterizes the indirect branch behavior in Java programs running in both interpreter and JIT mode. Compared to C programs, indirect branches in Java (either mode) are encountered more often, constitute a larger percentage of the dynamic branch count, and are more likely to have multiple targets. Interpreter mode execution results in more indirect branches and higher degrees of polymorphism than JIT mode. In addition, a small number of static indirect branches are found to account for a large percentage of indirect branch target mispredictions. For example, the 10 most critical polymorphic branches cause about three-fourths of the indirect branch-target mispredictions during Java execution.

The R-BTB copes with this behavior by identifying polymorphic branches that cause frequent mispredictions and rehashing their multiple targets into unified target storage. This is accomplished by augmenting a traditional target-storage structure with a target-history register for hashing, target-miss counters to identify critical branches, and a small critical indirect branch instruction buffer to store the critical polymorphic branches. The novelty of this scheme versus other indirect branch-target prediction schemes is that it does not split target storage resources between indirect and direct branches. Instead, it utilizes one large storage table and rehashes the targets of polymorphic branches into this table, allowing the resource allocation to be determined dynamically by usage.

The R-BTB eliminates 61% of indirect branch-target mispredictions caused by a traditional BTB for Java programs running in interpreter mode and eliminates 46% in JIT mode. Despite the possibility of introducing resource conflicts by rehashing, the overall branch-target misprediction rate is improved as well. Compared to a target cache with comparable resources, the R-BTB predicts

indirect branch targets more accurately for five out of six benchmarks. The R-BTB improves the overall branch prediction rate for four out of six benchmarks in interpreter mode and five out of six in JIT mode. The fact that R-BTB can benefit both Java execution modes also indicates potential performance improvement on other indirect branch intensive codes, such as C++, Perl, Tcl, etc.

APPENDIX

Table A1. Indirect Branch-Misprediction Rates on BTB, Target Cache, and R-BTB (Interpretation Mode)

Structures	Size (entry)		Indirect Misprediction Rate (%)					
			<i>db</i>	<i>jess</i>	<i>javac</i>	<i>jack</i>	<i>mtrt</i>	<i>compress</i>
BTB	2048		21.1	43.8	18.5	41.8	65.7	96.3
BTB + tagless target cache	1024 + 1024		13.7	21.1	13.1	24.9	18.7	29.4
	2048 + 512		18.1	26.8	18.3	32.7	21.8	38.7
	2048 + 1024		13.1	20.8	12.7	24.5	18.7	29.4
BTB + tagged target cache	4 way	1024 + 512	17	29.1	21	40.2	26.2	25.5
		1024 + 1024	11	20.4	15	28.7	21.7	13.2
		2048 + 256	28.7	42.6	29.7	55.6	32.8	39.5
		2048 + 512	16.6	28.9	20.8	39.9	26.1	25.5
		2048 + 1024	10.4	20.1	14.7	28.3	21.6	13.2
	16 way	1024 + 512	11.3	21.3	16.2	34.5	16.4	11.8
		1024 + 1024	8.3	14.2	12.4	22.7	14.2	9.6
		2048 + 256	17.3	34.9	22.7	44.9	22.5	25.3
		2048 + 512	10.8	21.1	15.9	34.3	16.4	11.8
		2048 + 1024	7.6	13.8	12	22.2	14.1	9.6
R-BTB	2048 + 16-entry, DM CIBIB		7.9	20.1	8.6	23.2	21	29.7

Table A2. Indirect Branch-Misprediction Rates on BTB, Target Cache, and R-BTB (Just-in-Time Compilation Mode)

Structures	Size (entry)		Indirect Misprediction Rate (%)					
			<i>db</i>	<i>jess</i>	<i>javac</i>	<i>jack</i>	<i>mtrt</i>	<i>compress</i>
BTB	2048		7.9	11.7	12.3	14.3	13.1	8.8
BTB + tagless target cache	1024 + 1024		9.9	10.8	12	10.4	8	4.8
	2048 + 512		13	13.6	15.2	14.2	11	4.9
	2048 + 1024		9.3	10	11.3	9.8	7.6	4.7
BTB + tagged target cache	4 way	1024 + 512	15.1	18.1	16.5	21.8	10.3	1.3
		1024 + 1024	10.8	13.7	12.5	13.8	6.7	1.1
		2048 + 256	25.1	26.6	23.2	34.8	16.3	2.6
		2048 + 512	14.7	17.6	16	21.2	9.9	1.3
		2048 + 1024	10.1	12.9	11.7	13	6.2	1.1
	16 way	1024 + 512	10.7	13	12.1	14.7	7.4	1.1
		1024 + 1024	8.2	10.2	9.6	8.4	4.3	1
		2048 + 256	16.4	19	17.1	27.4	11.8	1.4
		2048 + 512	10.2	12.5	11.7	14.1	7.1	1.1
		2048 + 1024	7.4	9.4	8.8	7.5	3.7	1
R-BTB	2048 + 16-entry, DM CIBIB		4.6	6.7	6.1	10	3.8	5.6

Table A3. All Branch-Misprediction Rates on BTB, Target Cache, and R-BTB (Interpretation Mode)

Structures	Size (entry)		All Branches Misprediction Rate(%)					
			<i>db</i>	<i>jess</i>	<i>javac</i>	<i>jack</i>	<i>mtrt</i>	<i>compress</i>
BTB	2048		3.2	9.8	3.4	10.6	14.8	35.2
BTB + tagless target cache	1024 + 1024		2.9	5.6	3	7	4.3	10.8
	2048 + 512		2.8	6.2	3.4	8.4	4.9	14.2
	2048 + 1024		2.2	4.8	2.5	6.3	4.2	10.8
BTB + tagged target cache	4 way	1024 + 512	3.3	7.4	4.3	10.8	6	9.4
		1024 + 1024	2.5	5.5	3.3	8	5	4.9
		2048 + 256	4.2	9.6	5.2	14.1	7.4	14.5
		2048 + 512	2.6	6.6	3.8	10.2	5.9	9.3
		2048 + 1024	1.8	4.7	2.8	7.2	4.9	4.8
	16 way	1024 + 512	2.6	5.7	3.5	9.4	3.8	4.4
		1024 + 1024	2.2	4.1	2.9	6.5	3.3	3.6
		2048 + 256	2.7	7.9	4.1	11.4	5.1	4.4
		2048 + 512	1.9	4.9	3	8.8	3.7	4.4
		2048 + 1024	1.4	3.3	2.3	5.7	3.2	3.5
R-BTB	2048 + 16-entry, DM CIBIB		1.7	5.3	2.1	7	5.5	11.7

Table A4. All Branch-Misprediction Rates on BTB, Target Cache, and R-BTB (Just-in-Time Compilation Mode)

Structures	Size (entry)		All Branches Misprediction Rate(%)					
			<i>db</i>	<i>jess</i>	<i>javac</i>	<i>jack</i>	<i>mtrt</i>	<i>compress</i>
BTB	2048		1.5	2.8	2.4	3.1	2.4	2.8
BTB + tagless target cache	1024 + 1024		2.2	3.4	3	3	1.7	1.6
	2048 + 512		2	3.1	2.8	3.1	2	1.6
	2048 + 1024		1.6	2.5	2.2	2.2	1.4	1.6
BTB + tagged target cache	4 way	1024 + 512	2.8	4.6	3.6	5.3	2.1	0.6
		1024 + 1024	2.3	3.8	3.1	3.6	1.5	0.5
		2048 + 256	3.4	5.2	3.9	7.4	2.9	0.9
		2048 + 512	2.2	3.7	2.9	4.6	1.8	0.5
		2048 + 1024	1.7	3	2.3	2.9	1.2	0.4
	16 way	1024 + 512	2.3	3.7	3	3.8	1.6	0.5
		1024 + 1024	2	3.3	2.7	2.5	1.0	0.5
		2048 + 256	2.4	4	3	5.8	2.2	0.5
		2048 + 512	1.7	2.9	2.3	3.1	1.3	0.4
		2048 + 1024	1.4	2.4	1.9	1.7	0.8	0.4
R-BTB	2048 + 16-entry, DM CIBIB		1.2	2.2	1.7	2.9	0.9	1.8

REFERENCES

- AIGNER, G. AND HÖLZLE, U. 1996. Eliminating virtual calls in C++ programs. In *Proceedings of 10th European Conference on Object Oriented Programming*. 142–166.
- CALDER, B., GRUNWALD, D., AND ZORN, B. 1994a. Quantifying behavioral differences between C and C++ programs. *J. Programming Languages*. 2, 4, 323–351.
- CALDER, B. AND GRUNWALD, D. 1994b. Reducing indirect function call overhead in C++ programs. In *Proceedings of the International Symposium on Principles of Programming Languages*. 397–408.
- CHANG, P. Y., HAO, E., AND PATT, Y. N. 1997. Target prediction for indirect jumps. In *Proceedings of the 24th International Symposium on Computer Architecture*. 274–283.
- DRIESEN, K. AND HÖLZLE, U. 1998a. Accurate indirect branch prediction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*. 167–178.

- DRIESEN, K. AND HÖLZLE, U. 1998b. The cascaded predictor: Economical and adaptive branch target prediction. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*. 249–258.
- DEAN, J., DEFOUW, G., GROVE, D., LITVINOV, V., AND CHAMBERS, C. 1996. Vortex: An optimizing compiler for object-oriented languages. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 83–100.
- ERTL, A. M. AND GREGG, D. 2001. The behavior of efficient virtual machine interpreters on modern architectures. In *Proceedings of Euro-Par 2001*. 403–412.
- ERTL, A. M. AND GREGG, D. 2003. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *Proceedings of SIGPLAN'03 Conference on Programming Language Design and Implementation*. 278–288.
- HSIEH, C. H. A., CONTE, M. T., JOHNSON, T. L., GYLLENHAAL, J. C., AND HWU, W. W. 1997. A study of the cache and branch performance issues with running java on current hardware platforms. In *Proceedings of COMPCON*. 211–216.
- KALAMATIANOS, J. AND KAEIL, D. R. A. 2000. Predicting indirect branches via data compression. In *Proceedings of the International Symposium on Microarchitecture*, 1998.
- LEE, J. AND SMITH, A. 1984. Branch prediction strategies and branch-target buffer design. *IEEE Computer* 17, 1, 1984.
- LI, T., JOHN, L. K., VIJAYKRISHNAN, N., SIVASUBRAMANIAM A., SABARINATHAN, J., AND MURTHY, A. 2000. Using complete system simulation to characterize SPECjvm98 benchmarks. In *Proceedings of ACM International Conference on Supercomputing*. 22–33.
- LI, T. AND JOHN, L. K. 2001. Understanding control-flow transfer and its predictability in java processing. In *Proceedings of the 2001 IEEE International Symposium on Performance Analysis of Systems and Software*. 65–76.
- LI, T., BHARGAVA R., AND JOHN, L. K. 2002. Rehashable BTB: An adaptive branch-target buffer to improve the target predictability of java code. In *Proceedings of the 9th International High Performance Computing Conference*. 597–608.
- LINDHOLM, T. AND YELLIN, F. 1999. *The Java Virtual Machine Specification*. 2nd Ed. Addison Wesley, Reading, MA.
- RADHAKRISHNAN, R., VIJAYKRISHNAN, N., JOHN, L. K., AND SIVASUBRAMANIAM, A. 2000. Architectural issue in java runtime systems. In *Proceedings of the 6th International Conference on High Performance Computer Architecture*. 387–398.
- REINMAN, G. AND JOUPPI, N. P. 2000. CACTI2.0: An integrated cache timing and power model. *WRL Research Report*, July, 2000.
- ROSENBLUM, M., HERROD, S. A., WITCHEL, E., AND GUPTA, A. 1995. Complete computer system simulation: The SimOS approach. *IEEE Parallel Distributed Technology: Systems Applications*, 3, 4, 34–43.
- SPEC JVM98 Benchmarks, <http://www.spec.org/osg/jvm98/>.
- VLAOVIC, S., DAVIDSON E. S., AND ATYSON, G. S. 2000. Improving BTB performance in the presence of DLLs. In *Proceedings of the International Symposium on Microarchitecture*. 77–86.
- VIJAYKRISHNAN, N. AND RANGANATHAN, N. 1999. Tuning branch predictors to support virtual method invocation in java. In *Proceedings of the 5th USENIX Conference of Object-Oriented Technologies and Systems*. 217–228.
- YEH, T. AND PATT, Y. 1995. A comparison of dynamic branch predictors that use two levels of branch history. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. 257–266.