Rehashable BTB: An Adaptive Branch Target Buffer to Improve the Target Predictability of Java Code

Tao Li, Ravi Bhargava and Lizy Kurian John

Laboratory for Computer Architecture
Department of Electrical and Computer Engineering
The University of Texas at Austin, Austin, TX 78712, USA
{tli3,ravib,ljohn}@ece.utexas.edu

Abstract. Java programs are increasing in popularity and prevalence on numerous platforms, including high-performance general-purpose processors. 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 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 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 target cache scheme for a majority of Java programs by adapting to a greater variety of indirect branch behaviors.

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. However, the Java runtime 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. The trend toward wide issue and deeply pipelined designs increases the penalty for mispredicting control transfer. Therefore, accurate control flow prediction is a critical performance issue on current and future microprocessors. Current processors predict branch targets with a branch target buffer (BTB), which caches the most recently resolved target [5]. Most indirect branches are unconditional

S. Sahni et al. (Eds.): HiPC 2002, LNCS 2552, pp. 597-608, 2002.

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. 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 [3]. 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 a clean, modular code design style. Virtual subroutines execute indirect branches using virtual method tables, which create additional indirect branches for most Java compilers.

Previous studies have concentrated mainly on the analysis and optimization of indirect branch prediction for SPEC integer and C++ programs [1][2][3]. Table 1 compares the indirect branch frequency found in Java processing with that found in the SPEC CINT95 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. On average, 20% of branches in Java are indirect branches while only 8% are indirect branches in the SPEC CINT95 C benchmarks. In addition, compared with C++ programs [3], the Java workloads studied here execute indirect branches more frequently.

Table 1. Indirect Branch Frequency in Java and C Programs[1]

Benchmarks		% of Indirect Branches in Instruction Stream	
		Interpretation	Just-in-Time Compilation
Java (SPEC JVM98)	*db*	3.0	2.5
	jess	3.3	2.4
	javac	2.6	1.9
	jack	2.5	2.1
	mtrt	2.7	2.0
	compress	4.3	1.6
C (SPEC CINT95)	*go*	0.7	
	compress	0.4	
	m88ksim	0.8	
	gcc	1.1	
	ijpeg	0.2	
	li	2.0	
	perl	2.2	
	vortex	0.9	

The frequency is the percentage of all instructions that are indirect branches, which includes all control transfer instructions. The indirect branch instruction mix ratios in Java programs are presented for runs in interpreter-only mode and JIT mode.

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, the 10 most critical indirect branches are responsible for 75% of indirect branch mispredictions (on average for the studied SPEC JVM98 benchmarks). Therefore, a solution that can effectively handle target prediction for a small number of polymorphic branch sites could improve the BTB performance.

[1] Sun JDK and SPECInt95 are compiled with MIPSpro C Compiler v7.3 with -O3 option. Simulations are performed on SimOS with IRIX5.3 OS. Initial instructions skipped: 1,000M; Instructions simulated: 200M. Input data set: S100 for SPECjvm98, ref for SPECInt95.

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 algorithm 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 [1]) 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 Rehashable BTB design. Section 5 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 6 discusses the related work. Finally, Section 7 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 Rehashable BTB scheme. To analyze the entire execution of the JVM and Java workloads, we use the SimOS full-system simulation framework [10] 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 [11] suite described in Table 2 is used for this research[2].

Table 2. SPEC JVM98 Benchmarks and their Indirect Branch Statistics

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

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 [6][7]. We simulate each benchmark on the SimOS MXS model until completion, except for the benchmark *compress* running in interpreter-only mode. In this case, we use the first 2,000M instructions. Table 2 reports the number of static and dynamic indirect

[2] We exclude the benchmark *mpegaudio* from our experiments because it failed to execute on the detailed model of SimOS.

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 in state-of-the-art Java technologies and on resource-constrained platforms, so we present analysis for both scenarios.

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 interpreter mode and JIT mode.

3.1. Polymorphic vs. 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 1 reports the percentage of dynamic indirect branches that are monomorphic (target=1) and polymorphic (targets>=2). The remaining bars illustrate the degree of polymorphism. In the interpreter mode, over 50% of the executed indirect branches are polymorphic, on average. 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 dynamic indirect branches are still polymorphic.

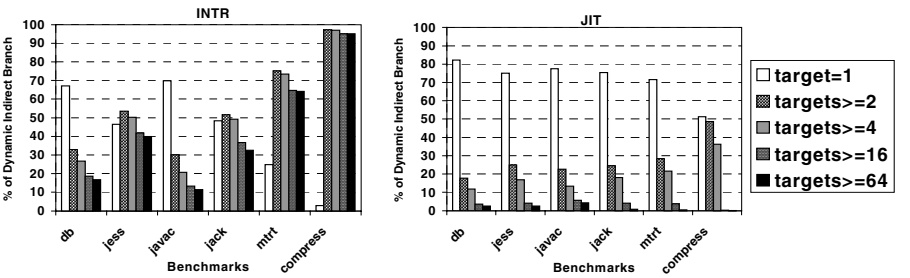

Fig. 1. Dynamic Indirect Branch Target Distribution

Although a large percentage of indirect branches are polymorphic, Figure 2 shows that a much smaller percentage of the static branches are polymorphic, 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.

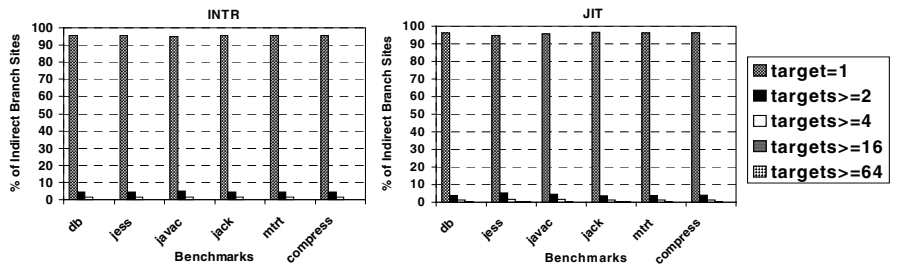

Fig. 2. Static Indirect Branch Site Target Distribution

3.2. The Impact of Polymorphic Indirect Branches

A few critical polymorphic indirect branches can deteriorate target prediction performance significantly. Figure 3 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. A study of the source code indicates that these polymorphic indirect branches come from code performing bytecode interpretation, calls to the dynamically shared native interface libraries and other JVM management routines. These critical polymorphic indirect branches show highly interleaved target transfer patterns, which cannot be predicted accurately with a conventional BTB structure [7].

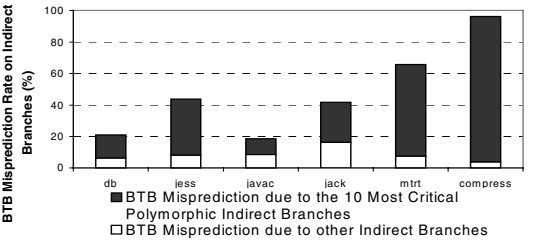

Fig. 3. Impact of Critical Polymorphic Indirect Branches

Benchmarks run in interpreter mode. BTB is 4-way with 2k entries

4. The Rehashable 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 [1].

4.1. Target Cache

The target cache scheme (shown in Figure 4) 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.

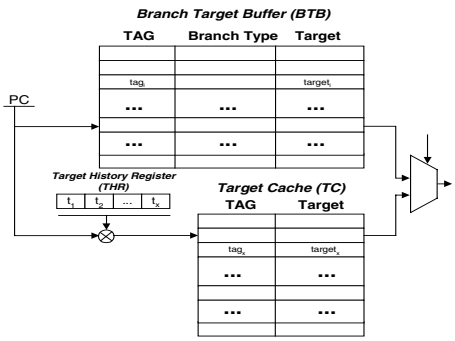

Fig. 4. Target Cache (TC) Scheme

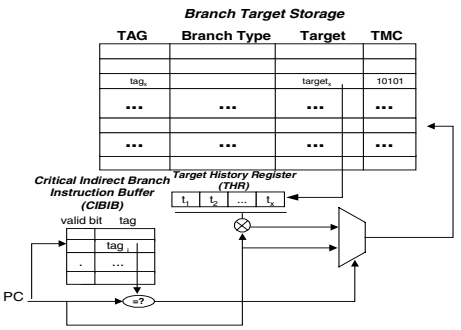

Fig. 5. Rehashable BTB (R-BTB)

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 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 Rehashable BTB (shown in Figure 5), which employs a small structure, the Critical Indirect Branch Instruction Buffer (CIBIB), to identify the performance-critical polymorphic indirect branches. 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 Rehashable 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 by Figure 5, a CIBIB entry consists of a tag 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. For example, for a 2048-entry, four-way R-BTB, a 11-bit index is generated. The most significant 9 bits are used to choose among the 512 sets and the lower two bits choose among the four ways.

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 5), 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. In this work, a global THR is used.

4.3. Target Prediction with the R-BTB

This section provides a detailed example of target prediction using the Rehashable 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 a R-BTB entry index. If the branch PC misses in the CIBIB, it is used to index the R-BTB (Figure 6.a).

At runtime, the PCs of critical indirect branches with a high target misprediction rate are dynamically identified, removed from the branch target storage, and sent to the CIBIB (Figure 6.b). When a branch PC hits in the CIBIB, the target address is found in the rehashed entry of the R-BTB. This entry is located by XORing the branch PC with the THR value. By using the target history pattern as a hashing input, the multiple targets of critical polymorphic indirect branches are stored in different entries of the R-BTB (Figure 6.c). In this manner, the R-BTB houses targets of both indirect and direct branches.

5. 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.

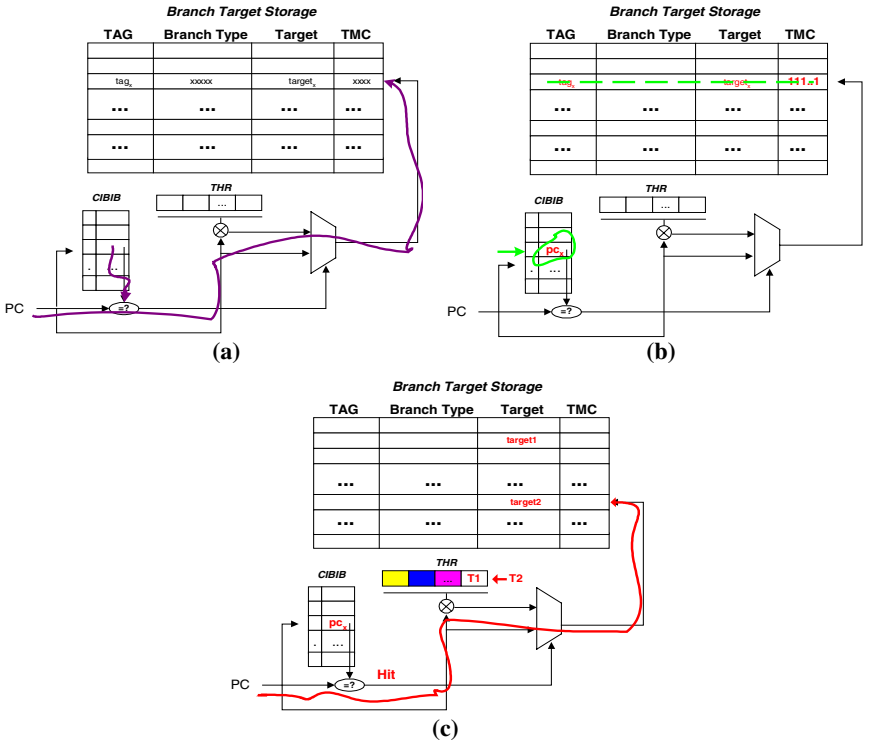

Fig. 6. Target Prediction with R-BTB

5.1 Evaluated Target Predictors

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 [1][2] 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 5.4.

The target predictors in this section are also used to predict branch targets for branch types other than indirect branches. In addition to indirect branches, taken conditional branches access the target predictors in our evaluation. Although it depends on the architecture, target prediction is not always necessary for predicting fall-through paths of not-taken branches or for predicting direct branch targets.

5.2 Branch Target Prediction Performance

Tables 3 and 4 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 results in 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.

Table 3. Indirect Branch Target Misprediction Rates (Interpreter Mode)

Structures	Size (# of entries)	Indirect Misprediction Rate (%)					
		db	*jess*	*javac*	*jack*	*mtrt*	*compress*
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 4. Indirect Branch Target Misprediction Rates (JIT Compilation Mode)

Structures	Size (# of entries)	Indirect Misprediction Rate (%)					
		db	*jess*	*javac*	*jack*	*mtrt*	*compress*
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 5. Overall Branch Target Misprediction Rates (Interpreter Mode)

Structures	Size (# of entries)	Overall Branches Misprediction Rate (%)					
		db	*jess*	*javac*	*jack*	*mtrt*	*compress*
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

Table 6. Overall Branch Target Misprediction Rates (JIT Compilation Mode)

Structures	Size (# of entries)	Overall Branches Misprediction Rate (%)					
		db	*jess*	*javac*	*jack*	*mtrt*	*compress*
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

Improving indirect branch target prediction performance can only benefit the processor if the overall branch target prediction performance is also improved. Tables 5 and 6 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 the reduces the overall branch target 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.

5.3 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, then the R-BTB behaves like a traditional BTB. When the number of polymorphic branches is high, then the R-BTB 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 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 and this increases the chance for entry pollution or corruption. This is the scenario where smaller, dedicated indirect branch target storage proves beneficial. However, previous work [7] indicates that due to high target locality and a low number of polymorphic branch sites, the BTB corruption caused by rehashing and reuse of an already allocated BTB entry should be low.

5.4 Dynamic Target Storage Allocation versus Static Target Allocation

Previous work with the target cache [1] splits resources differently than 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 7 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 [1]), 2048+1024, and 2048+2048. In addition to the four-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.

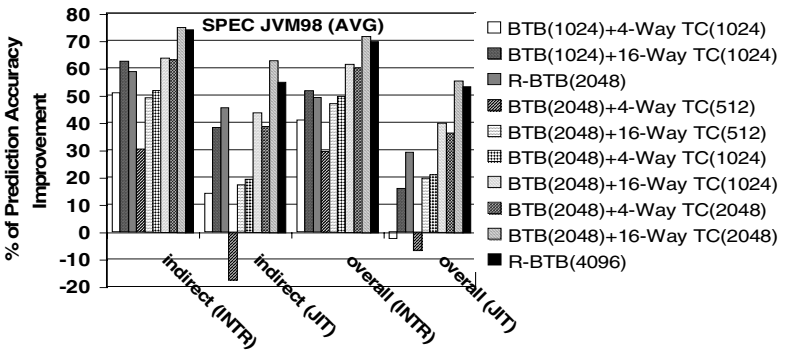

Fig. 7. Impact of Target Cache Resource Partitioning

Performance is with respect to a 4-way, 2048-entry BTB. TC stands for target cache

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 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. Related Work

Lee and Smith [5] 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. Hsieh et al. [4] 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. A related study [12] 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 et al. [7] characterized control flow transfer in Java processing using full-system simulation and SPEC JVM98 benchmarks. However, no hardware optimization was proposed.

Driesen and Hölzle [2] 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 also proposed a cascaded predictor [3], 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 more strict filtering criteria 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.

7. 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. The novelty of this scheme versus other indirect branch target prediction schemes is that it does not split target storage resources between indirect branches and direct branches. Instead, it utilizes one large storage table and rehashes the targets of polymorphic 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.

Acknowledgement

Ravi Bhargava is currently supported by an Intel Foundation Graduate Fellowship Award. This research is supported in part by the National Science Foundation under grant numbers 0113105 and 9807112, by a State of Texas Advanced Technology Program grant, and by Tivoli, Motorola, Intel, IBM and Microsoft Corporations.

References

[1] P. Y. Chang, E. H. and Y. N. Patt, Target Prediction for Indirect Jumps, In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 274-283, 1997

[2] K. Driesen and U. Hölzle, Accurate Indirect Branch Prediction, In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 167-178, 1998

[3] K. Driesen, and U. Hölzle, The Cascaded Predictor: Economical and Adaptive Branch Target Prediction, In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 249-258, 1998

[4] C. H. A. Hsieh, M. T. Conte, T. L. Johnson, J. C. Gyllenhaal and W. W. Hwu, A Study of the Cache and Branch Performance Issues with Running Java on Current Hardware Platforms, In *Proceedings of COMPCON*, pages 211-216, 1997

[5] J. Lee and A. Smith, Branch Prediction Strategies and Branch Target Buffer Design, *IEEE Computer 17(1)*, 1984

[6] T. Li, L. K. John, N.Vijaykrishnan, A. Sivasubramaniam, J. Sabarinathan and A.Murthy, Using Complete System Simulation to Characterize SPECjvm98 Benchmarks, *In Proceedings of ACM International Conference on Supercomputing,* pages 22-33, 2000

[7] T. Li and L. K. John, 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*, pages. 65-76, 2001

[8] T. Lindholm and F. Yellin, The Java Virtual Machine Specification, Second Edition, Addison Wesley, 1999

[9] R. Radhakrishnan, N. Vijaykrishnan, L. K. John and A. Sivasubramaniam, Architectural Issue in Java Runtime Systems, In *Proceedings of the 6th International Conference on High Performance Computer Architecture*, pages 387-398, 2000

[10] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta, Complete Computer System Simulation: the SimOS Approach, *IEEE Parallel and Distributed Technology: Systems and Applications*, vol.3, no.4, pages 34-43, Winter 1995

[11] SPEC JVM98 Benchmarks, http://www.spec.org/osg/jvm98/

[12] N. Vijaykrishnan and N. Ranganathan, Tuning Branch Predictors to Support Virtual Method Invocation in Java, In *Proceedings of the 5th USENIX Conference of Object-Oriented Technologies and Systems*, pages. 217-228, 1999