Modeling and Evaluation of Control Flow Prediction Schemes Using Complete System Simulation and Java Workloads

Tao Li, Lizy Kurian John and Robert H. Bell, Jr
Department of Electrical and Computer Engineering
The University of Texas at Austin, Austin, TX 78712
{tli3,ljohn,belljr}@ece.utexas.edu

Abstract

Program control flow transfer (branch) prediction is considered to be a performance hurdle and a key design issue for current and future microprocessors. Branch prediction schemes with high prediction accuracy have been proposed to support longer processor pipelines with higher frequency clocks. In the previously published literature, the design and evaluation of branch predictors have been based heavily on the simulation of only user instructions from scientific and commercial workloads written in programming languages such as C or C++. To complement the existing research, this paper presents a case study of the modeling and evaluation of advanced branch predictors using full-system simulation of Java workloads running on a commercial operating system.

The contributions of this paper are: (1) The presentation of a full system simulation framework to model, simulate and evaluate the performance of a set of advanced prediction schemes on emerging Java workloads; (2) An analysis of the performance and design complexity of advanced branch predictors in the presence of full system code; (3) An accurate modeling of user/kernel branch aliasing on a wide range of branch predictors.

1. Introduction

The current generation of microprocessors uses wide instruction issue and out-of-order execution to exploit the inherent *instruction level parallelism* (ILP) in programs to improve performance. A processor attempts to issue multiple decoded instructions to the execution units every cycle in parallel. To reduce the possibility of pipeline stalls, the instruction fetch unit must match the issue rate and continuously provide instructions to the decoder. However, in the case of a control flow transfer, there may be many cycles of latency between the fetch of the branch instruction and the execution and resolution of the branch direction and target. Without knowing whether the branch is taken or not, the fetch unit must stall or execute subsequent instructions speculatively until the branch is executed and the next instruction in program order is determined.

This research is supported 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.

Branch prediction is a technique that supports the execution of speculative instructions that are more often the correct instructions in program order. Branch prediction hardware stores and uses past branch behavior to predict the branch direction accurately. This allows processors to exploit program ILP correctly more often, allowing more work to be accomplished in less time and a corresponding enhancement of performance. However, even with specialized hardware the prediction is sometimes not correct, and the speculative instructions previously thought to be next in program order must be discarded when the branch is resolved. The accuracy of the control flow predictor is therefore considered to be a performance hurdle and a key design issue for current and future microprocessors (e.g. the Intel *Pentium-4*) in which the latency between branch prediction and resolution, the misprediction penalty, is high.

Several highly accurate branch predictors have been proposed in the literature [15][7][2][14][4]. In those studies, the specific design of the branch predictors and the performance evaluation relied heavily on the simulation of user-mode instructions using traditional workloads (e.g. *SPECint* written in C).

The combined impact of kernel code and user code on overall branch predictor performance has received limited attention. The effects of operating system context switches were modeled in user-only simulation studies by flushing branch prediction tables at regular intervals [8][2]. However, periodic flushing was found to estimate user/kernel branch interactions only poorly, and user-mode instructions by themselves were shown to provide accurate prediction results only when the kernel accounted for less than 5% of the total executed instructions [3]. Similarly, there have been few studies that focused on the predictability of emerging workloads like Java [9][6] despite the fact that Java has been shown to have a significant proportion of kernel-mode branches and to exercise a large number of branch sites compared to benchmarks such as *SPECint95* [5][6]. This makes Java code more susceptible to a reduction in performance due to negative interactions in the branch predictor (branch aliasing) between user code and kernel code. Conclusions in those papers were based on limited simulations of simple branch predictors and did not take into account the ability of advanced predictors to reduce aliasing.

To help quantify these effects, this paper presents a case study of the modeling and evaluation of advanced branch predictors using full-system simulation of Java workloads running on a commercial operating system. The rest of this paper is organized as follows. Section 2 provides a brief

Proceedings of the 10th IEEE Int'l Symp. on Modeling, Analysis, & Simulation of Computer & Telecommunications Systems (MASCOTS'02)
1526-7539/02 $17.00 © 2002 **IEEE**

background on the branch prediction schemes examined in this study. Section 3 describes the simulation methodology and experimental setup. Section 4 discusses the experimental results. Section 5 models the user/kernel branch aliasing. Finally, section 6 presents some conclusions.

2. Background

In this section, we give a brief introduction to dynamic branch predictors and then describe the set of recently proposed, state-of-the-art predictors that are used in this study.

2.1 Dynamic Branch Predictors

Figure 1 shows a generalized branch predictor structure. A branch predictor predicts the future direction (taken or not taken) of a branch by performing a lookup of the *Branch History Table* (BHT), which can be a $[2^k, 2^j]$ finite-state machine array implemented with 2-bit, up-down saturating counters. The 2-bit counter encodes the tendency of a branch to be taken or not taken in the past. It is updated when branches are executed and the branch direction is resolved.

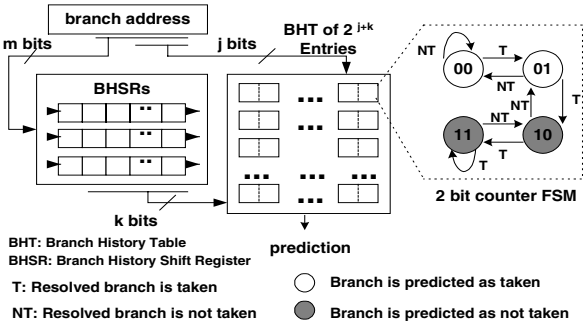

Figure 1. Dynamic Branch Predictor

The method of selection of a particular 2-bit counter in the BHT differentiates most branch predictor schemes. The simplest predictor, *2bc*, chooses a 2-bit counter from a one-dimensional BHT using a subset of the address bits of the branch. In the more sophisticated two-level dynamic branch predictors, a 2-bit counter is selected by row and column [15]. Bits from the branch address select the column, while the row is chosen using information from the *Branch History Shift Registers* (BHSRs), which collect the predicted directions of previous branches. The BHSR may keep separate local history buffers for different branch sites or it may use a single global history buffer for all branches. Global BHSR predictors exploit correlation in the sequence of branch predictions in a program. Correlation occurs when the past prediction sequence for a branch matches the value in the BHSR. Branches with different execution sequences are therefore differentiated, leading to better predictions. Schemes that use multiple local BHSRs map a set of branches to a BHSR using particular address bits, as shown in Figure 1. This exploits any repeating patterns in the execution of a single or a set of program branches (e.g. loop branches).

Table 1 summarizes the configurations of several dynamic predictors. The scheme name is derived from the taxonomy proposed in [15]. The size is normalized with respect to the hardware cost to implement a 2-bit counter. The variable i is the $\log_2$ of the number of BHT entries.

The BHT and BHSRs are updated dynamically at run time and the updates are used for future branch prediction. The fixed sizes of branch predictor tables, constrained by chip die area and access time, however, make it impossible to hold all dynamic branch information. Consequently, several branches map to the same entry in the prediction tables, a phenomenon known as *branch aliasing*. Previous studies show that branch aliasing is more likely to be destructive as prediction accuracy decreases [3][12]. In the following sections *Gshare* and the advanced de-aliasing branch predictors used in this study are described, and their specific features to reduce aliasing are discussed.

Table 1. Dynamic Branch Predictor Configurations[1]

Scheme Name	Category	Branch (PC) bits used for: BHSRs selection (m)	Branch (PC) bits used for: BHT column index (j)	BHSRs bits used for BHT row index (k)	Size of scheme (# of BHT entries)
2bc	1-level	0	i+10	0	2^iK
GAg	2-level, global BHSR	0	0	i+10	2^iK
GAs		0	i+6	4	2^iK
Gshare		0	0	i+10	2^iK
SAg	2-level, local BHSRs	i+8	$i+8-\log_2(i+9)$	i+9	2^iK
SAs		i+8	i+5	4	2^iK

2.2 Advanced Branch Predictors

Gshare is shown in Figure 2 [7]. To address the aliasing problem, *Gshare* uses the exclusive-or (XOR) of a global BHSR with the low-order address bits of a branch to form a BHT index. A fixed size global BHT in combination with the XOR function permits *Gshare* to use a longer prediction history, resulting in one of the best one-dimensional BHT predictors.

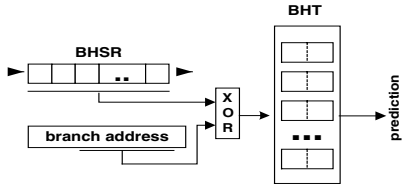

Figure 2. Gshare

Research studies show that the use of a global BHSR to select from a row of 2-bit counters is an effective branch prediction strategy for most of the integer benchmarks [15]. This is due in part to the abundant *if-else* instructions in integer programs that are often highly correlated. However,

[1] We omit the small hardware costs of the global BHSR on the *GAg*, *Gshare* and *GAs* schemes. We assume that two BHSR bits have the same hardware cost as one 2-bit counter for the *SAg* and *SAs* schemes.

destructive branch aliasing can still seriously deteriorate its performance [16][12][3]. As a result, more complex predictors have been proposed to alleviate aliasing effects and improve prediction accuracy. These are described below.

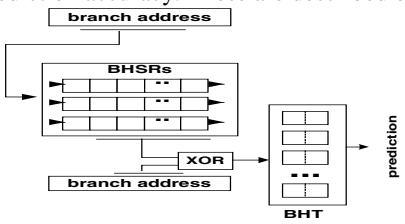

Figure 3. Pshare

Pshare, shown in Figure 3, is the per-set version of the *Gshare* predictor [2]. Instead of a single global BHSR, branch set addresses select one of the multiple BHSRs to choose a BHT entry. Like *Gshare*, the selected set BHSR is XORed with branch address bits to index a BHT entry. For our study, we implement 2K-entry BHSRs, as suggested in [2].

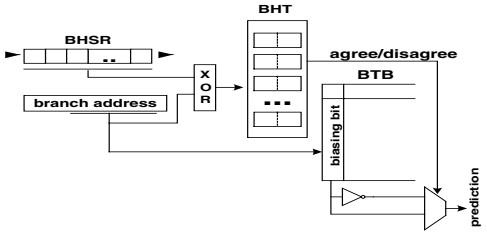

Figure 4. Agree

The *Agree* predictor, shown in Figure 4, converts instances of destructive aliasing into either constructive or neutral aliasing by attaching a biasing bit to each branch that predicts the most likely resolution of that branch [14]. The 2-bit BHT counter indicates whether or not the branch will go in the direction given by the biasing bit. The idea is that most branches that map to the same BHT entry but different *Branch Target Buffer* (BTB) entries are highly biased in one direction, i.e. they are always taken or always not taken. The BTB biasing bit captures the correct direction for the separate branches mapped to the same BHT, and the separate branches update the counter in the same direction when they agree with their biasing bit even if the directions are different. This reduces mispredictions for branches that would otherwise update the counter in opposite directions.

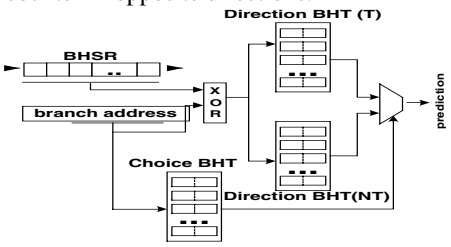

Figure 5. Bi-Mode

In the *Agree* predictor, the biasing bit is determined by the direction of the branch when it is initially introduced into the BTB. The *Bi-Mode* predictor, shown in Figure 5, uses a dedicated 'choice' BHT to dynamically determine the taken or not taken bias [4]. It uses two 'direction' BHTs of equivalent size, one for the taken direction and one for the not taken direction. When a branch is encountered, both 'direction' BHTs make predictions and the 'choice' BHT entry accessed by bits in the branch address makes the actual prediction. At resolution, only the 'direction' BHT chosen by the 'choice' BHT is updated. As a result of this scheme, branch predictions stored in a 'direction' BHT will have the same bias. Thus this classification helps to alleviate destructive aliasing while keeping the harmless aliasing together.

The *Multi-Hybrid* predictor, shown in Figure 6, instantiates more than two distinct predictors and uses a predictor selection counter to keep track of the most accurate component predictor for each branch [2]. This is a generalization of the classic two-component hybrid predictors [7]. A priority encoding mechanism is used to select the appropriate prediction. Using predictors with short training times (e.g., a simple static, *always-taken* predictor and *2bc*) to assist the otherwise more accurate predictors (e.g., *Gshare* and *GAs*) during their warm-up phases, *Multi-Hybrid* maintains high prediction accuracy even after a loss or intermingling of branch histories due to context switches.

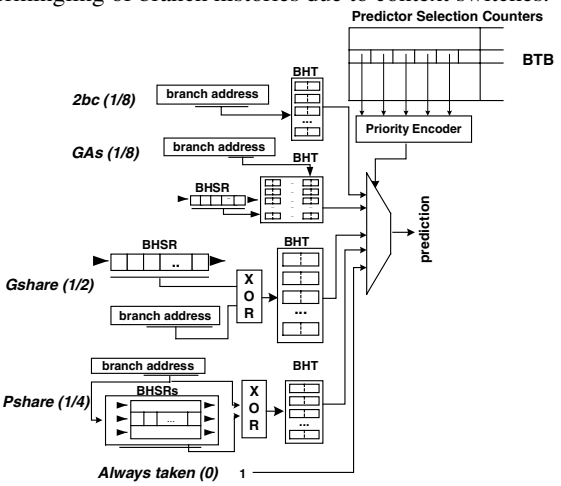

Figure 6. Multi-Hybrid

Note that our simulated *Multi-Hybrid* does not include an *AVG predictor* [2] because it requires source recompilation to provide profile information for accurate prediction of regular loop branches, which is not as applicable to more complicated software like an operating system or Java JVMs [1]. As suggested by [2], we allocate approximately half of the total area budget for *Gshare*, a quarter of the total budget for *Pshare*, and one-eighth each for *2bc* and *GAs*. The priority ordering of the component predictors is *2bc*, *GAs*, *Gshare*, *Pshare* and the simple *always taken* scheme.

All of the advanced predictor schemes examined in this study consist of a *Gshare* predictor component and other resources necessary for branch de-aliasing. These include the multiple component predictors and the predictor selection counter table for *Multi-Hybrid*, the biasing bit table for *Agree*, and the 'choice' BHT for the *Bi-Mode* predictor. Table 2 summarizes the normalized size of these advanced predictors as the size of their *Gshare* component is varied from 8K to 256K.

Table 2. Hardware Complexity of Advanced Predictors

Predictor	Additional Branch De-aliasing Hardware	Predictor Size Normalized to *Gshare*					
		8k	16k	32k	64k	128k	256k
Gshare	0	1	1	1	1	1	1
Pshare	2K BHSRs	2.63	1.88	1.47	1.25	1.13	1.07
Agree	2K biasing bits in BTB	1.13	1.06	1.03	1.02	1.01	1
Multi-Hybrid	additional single-scheme predictors, 5×2K predictor selection counters in BTB	3.25	2.63	2.31	2.16	2.08	2.04
Bi-Mode	additional 'direction' BHT and a 'choice' BHT of equivalent size	3	3	3	3	3	3

3. Simulation Framework and Experimental Setup

We use simulation of a complete system, including the operating system and user code, to model and evaluate the various branch prediction schemes discussed in Section 2. This section describes the simulation framework, the experimental setup, and the benchmarks that are used.

3.1 Simulation Framework

We use a full system simulation environment called *SimOS* [11] to execute Java workloads in the *Java Runtime Environment*, which in turn runs on top of a commercial operating system. *SimOS* models the major system hardware components including the CPU, the cache and memory system hierarchy, and the various I/O devices. These components are modeled in enough detail to boot and run the *Silicon Graphics IRIX5.3* operating system. *SimOS* supports multiple yet compatible CPU simulators including *Embra*, *Mipsy* and *MXS*. In this study, the fastest CPU simulator, *Embra*, is used to boot the operating system, mount the simulated workloads disk, and position the workloads for detailed investigation. Subsequently, each workload is executed at least once using the *Mipsy* simulator to warm up the file system and cache hierarchy. Finally, an individual starting checkpoint is generated for each workload. The checkpoint mechanism enables efficient simulation of different predictor schemes and configurations starting from an initial hardware state.

All instruction and data accesses for both applications and the operating system are modeled. A heavily instrumented version of the *SimOS MXS* [11] simulator is used to generate address traces that are then fed into the branch predictor simulators. The branch predictor configurations and simulators have been used for prior research and are described in [5][6]. Each predictor uses a 2K-entry, 4-way set-associative BTB for the branch target prediction. Full system simulation enables the collection of accurate data for predictor simulations without special-purpose hardware or intrusive modifications to the software.

3.2 Benchmarks

In this study, a Java Development Kit from *Sun Microsystems* and the *SPECjvm98* benchmarks are simulated on the top of the *SGI IRIX 5.3* operating system. *SPECjvm98* is a benchmark suite released by the *Standard Performance Evaluation Corporation* (SPEC) to evaluate performance for the combined hardware and software aspects of Java client technologies [13].

Table 3 gives a description of the *SPECjvm98* benchmarks and presents some general statistics for each benchmark. The ability of *SimOS* to run unmodified applications makes porting of the Java runtime system and execution of *SPECjvm98* straightforward. Each benchmark is executed from the *SimOS* command line, and a *Just-In-Time* (JIT) compiler [1] is used to translate Java byte-codes to native instructions at run time. We set additional annotations in the *SimOS* to ensure that each run is complete and accurate.

Table 3. SPECjvm98 Benchmarks[2] and Statistics

Benchmarks	Description	Conditional Branch Statistics		% of OS cycles	Benchmark Category
		Static Sites	Dynamic Instances		
db	Performs multiple database functions on a memory resident database	39,973	32,890,218	31	*G-I*
jess	Java expert shell system based on NASA's CLIPS expert system	44,691	64,252,325	30	
javac	The JDK 1.0.2 Java compiler compiling 225,000 lines of code	44,885	55,573,959	19	*G-II*
jack	Parser generator with lexical analysis, early version of what is now JavaCC	46,782	251,173,727	17	
mtrt	Dual-threaded raytracer	42,728	219,017,400	7	*G-III*
compress	Modified Lempel-Ziv method (LZW) to compress and decompress large file	39,988	432,529,058	6	

[2] The benchmark *mpegaudio* is excluded from our experiments because it did not run on the detailed CPU model of *SimOS*.

Compared with *SPECint95* benchmark statistics described in [6], Java applications contain a much larger number of static branches. As shown in Table 3, there is a wide variability in the amount of time different workloads spend in user and kernel code, with operating system activity ranging from 6% in *compress* to 31% in *db*. This observation suggests that emerging workloads such as Java can exhibit a stronger operating system performance component than do SPEC integer workloads, which in turn motivates the use of full system simulation to evaluate the performance of branch prediction schemes. To ease experimental analysis, the benchmarks are partitioned into three groups (G-I, G-II and G-III) based on their degree of operating system activity, as shown in Table 3.

4. Branch Predictor Simulation Results

In this section, full-system simulation experiments are used to find out: (1) whether the advanced branch predictors benefit both user and OS code; (2) whether the best prediction schemes for user code are also the best schemes for both user and kernel code; and (3) what the most cost-effective prediction schemes are for user and kernel code.

4.1 Misprediction Rates on Advanced Branch Predictors

Figure 7 shows the prediction accuracies (represented as misprediction rates) of the prediction schemes with respect to the size implied by the *Gshare* component in each predictor. As described in Section 2, advanced branch predictors consume more hardware resources than the simpler *Gshare* scheme, due to the additional hardware overhead used to reduce aliasing. For example, the *Multi-Hybrid*, *Agree* and *Bi-Mode* predictors with a 64K *Gshare* component actually contain 138K (128K+10K), 66K (64K+2K) and 192K (128K+64K) BHT entries, respectively. We present the average statistics of the two benchmarks in each G-I, G-II and G-III group unless especially specified. To gain more insight into the performance of the predictors in a complete system, we also break out the misprediction rates into user and operating system (or kernel) components.

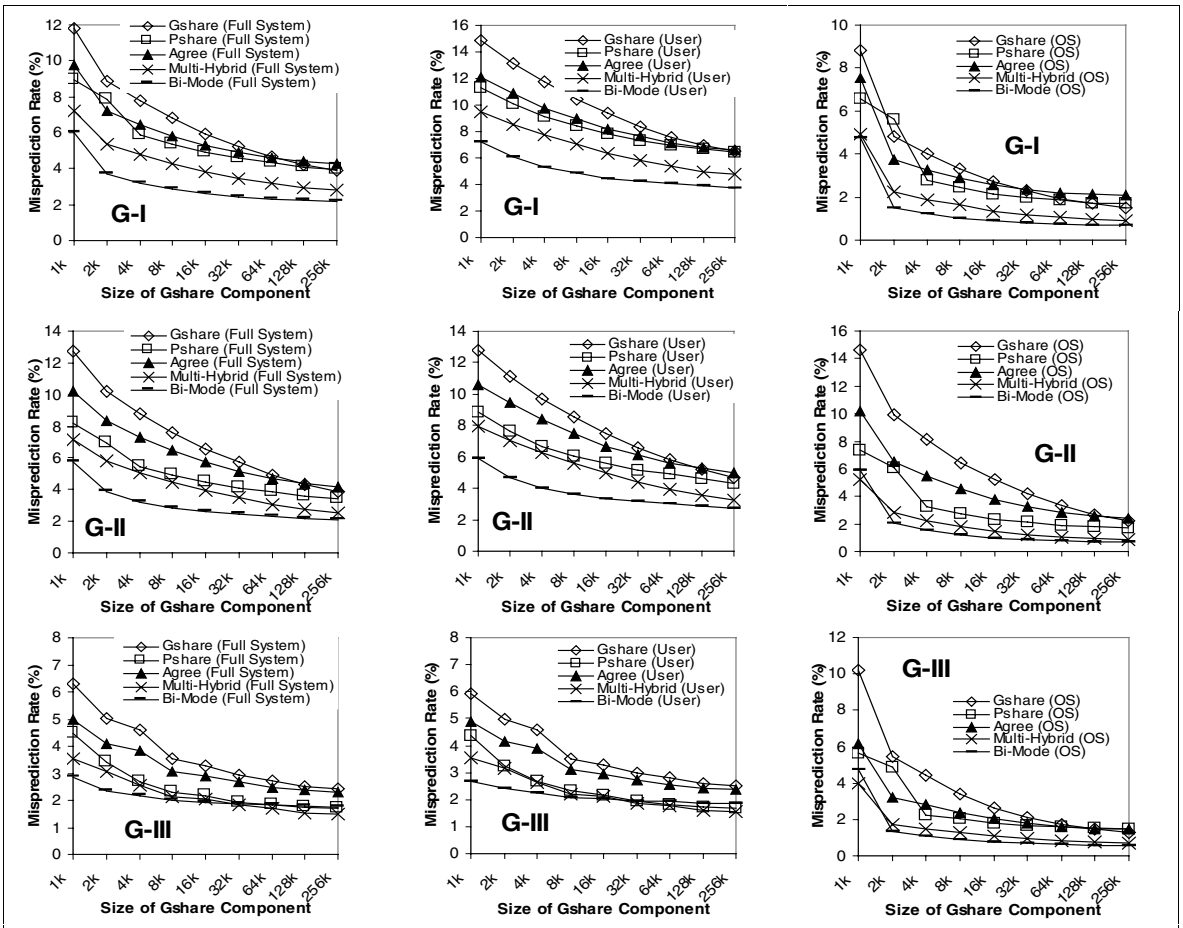

Figure 7. Misprediction Rates of Advanced De-Aliasing Predictors for User, OS and Full-system Code

Figure 7 leads to a number of interesting observations. In general, advanced de-aliasing techniques benefit both user and operating system code. For example, on predictors with a *Gshare* component of 16K BHT entries, the techniques used in *Pshare*, *Agree*, *Multi-Hybrid* and *Bi-Mode* predictors reduce the misprediction rates on the G-I benchmarks from 2.71% to 2.16%, 2.57%, 1.34% and 0.89% on the operating system code. The percentages of misprediction reductions are 20%, 5%, 51% and 67%, respectively. For user code, the corresponding misprediction reduction percentages are 16%, 12%, 32% and 52%, implying that the techniques benefit operating system code more than user code.

However, the advanced de-aliasing techniques do not always outperform the simple *Gshare* scheme. For example, for operating system code, *Pshare* and *Agree* are found to yield higher misprediction rates for the G-I benchmarks than *Gshare* in some cases. Using a single BHSR, the *Gshare* predictor exploits global branch correlation in operating system code in which binary decision tree-based branching sequences occur frequently. Using local BHSRs, *Pshare* can perform worse for operating system code because it does not use the full global branch correlation. In the *Agree* predictor, if the branch does not show strongly biased behavior, there is still frequent aliasing between instances of a branch that agree with the biasing bit and instances which do not agree with the biasing bit. Furthermore, if the biasing bit, which is chosen the first time the branch is introduced into the BTB, turns out to be not optimal for performance, it remains in the BTB until eviction by a different branch. The non-optimal biasing bit can pollute the BHT with 'disagree' results. The *Multi-Hybrid* and *Bi-Mode* predictors tend to achieve higher overall accuracies across benchmarks that exhibit wide variations in user and kernel activity. Of course, the price for the increased consistency in performance is the application of additional hardware resources relative to other schemes.

Also shown in Figure 7, the best dynamic prediction scheme for user code is always the same as the best scheme for the operating system code and also the best scheme for both user and kernel code in a full system. This occurs despite negative aliasing in the branch predictor caused by user and operating system code interaction. Section 5 investigates the effects of user/kernel aliasing in more detail.

For benchmarks in groups G-II and G-III, the misprediction rates in user code give a good approximation of the overall prediction accuracy, as expected from [3]. However, the aggregate prediction accuracies on G-I benchmarks with a relatively high operating system activity component are not well modeled by user instruction misprediction rates. For example, *Multi-Hybrid* at a 64K *Gshare* component achieves a user misprediction rate of 5.36% while the full-system misprediction rate is 3.17%. Despite the fact that the operating system code is highly complicated by both hardware and software management activities, the kernel code shows higher branch prediction accuracies than the user code. This observation is consistent with the results reported in [10], in which the branch predictability of an operating system running an *Apache* web server benchmark is analyzed.

4.2 Performance and Complexity Tradeoffs of Advanced Branch Predictors

As shown in Figure 7, advanced de-aliasing techniques enhance predictor accuracies across benchmarks with varying degrees of user and kernel activity at the cost of additional hardware and complexity. This introduces the interesting question of whether the performance improvement justifies the increased design complexity. Due to their larger sizes, advanced branch predictors have increased access latencies and energy consumption.

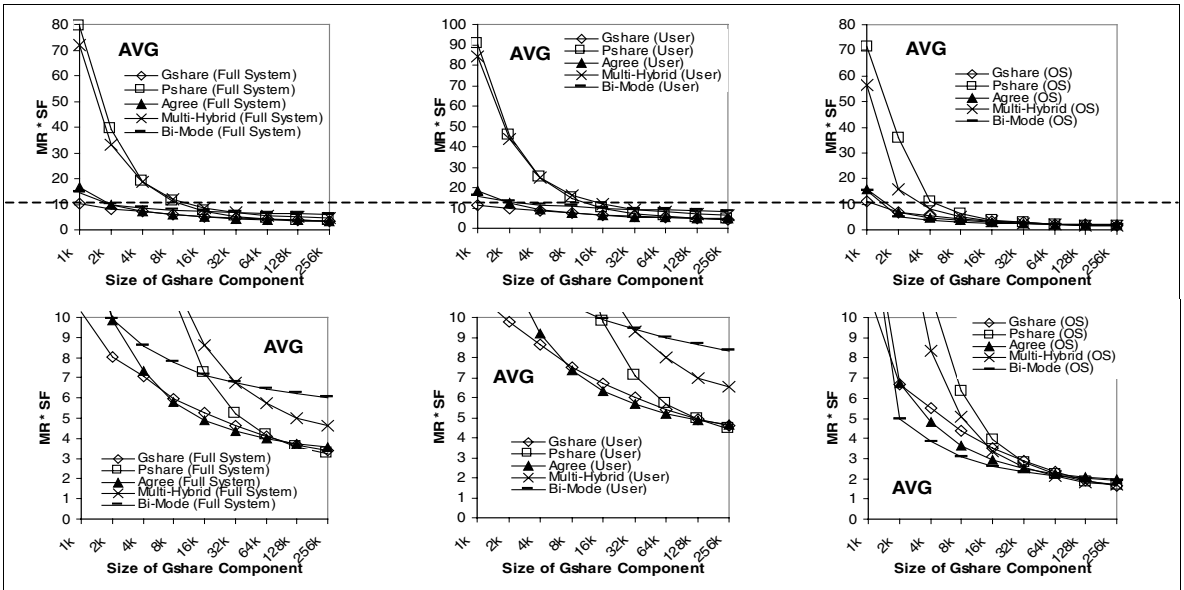

Figure 8.Performance/Complexity of Advanced De-Aliasing Techniques (Bottom graphs enlarge the portion of the top graphs below the dashed line)

To quantify the performance and complexity tradeoffs, we define the *Size Factor* (*SF*) of the branch predictor as its actual size normalized to the size of its corresponding *Gshare* component. We then use the product of the misprediction rate and the size factor (*MR*SF*) as a metric that accounts for both performance and complexity. Figure 8 shows the results (average statistics) of all studied benchmarks.

Comparing the results in Figures 7 and 8, it is immediately observed that the most accurate prediction scheme is not always the most cost-effective. For example, the *Agree* predictor in Figure 7 yields higher misprediction rates than those of *Multi-Hybrid* and *Bi-Mode*. However, when both performance and size issues are considered, the *Agree* predictor proves to be a cost-effective design choice. Figure 8 also shows that the optimal performance/complexity predictors are different for user and operating system code. For example, the *Bi-Mode* scheme is the most cost-effective prediction scheme for medium or large size (4K-256K) operating system code branch predictors. Even so, it is less efficient for user code. Summarizing Figure 8, the *Gshare* and *Agree* schemes are the most efficient branch predictors for the full system when both accuracy and predictor size are considered.

5. Modeling User/Kernel Branch Aliasing

In this section, we model user/kernel branch aliasing. As described earlier, aliasing occurs when different branch sites map to the same entry of prediction hardware structures such as the BHT. Note that all advanced de-aliasing predictors use the same *Gshare* predictor index scheme. To examine a wide range of predictor index schemes, we model all predictors shown in Table 1 in our experiment.

5.1 User/Kernel Branch Aliasing in BHT

We instrument our branch prediction simulators to record the histograms of mappings between branch instructions and BHT entries. We capture events both per-BHT entry and per-BHT access reference, and we mark each as a member of one of the following categories: (1) if a BHT entry or access is mapped to the same branch site from user (or kernel) space, we record an instance of a *user/user* (or a *kernel/kernel*) *hit*; (2) if a BHT entry or access is mapped to different branch sites from user (or kernel) space, we record *user/user* (or *kernel/kernel*) *aliasing*; (3) if a BHT entry or access is mapped to branches from different spaces, we record *user/kernel aliasing*; (4) if a BHT entry is never mapped during the simulation, we treat it as *unmapped*. Finally, a *cold miss* is counted when a BHT entry is first mapped to a branch. To reduce the effect of capacity misses, we examine the mapping behavior on a BHT with 8K entries.

The results for benchmark groups G-I, G-II and G-III are shown in Figures 9 and 10. In a processor with fine-grained resource sharing like a superscalar machine, the presence of operating system branches changes the utilization of microarchitecture resources like the BHT. For example, for the G-I benchmarks, operating system branches occupy 11% (*SAg* predictor) to 22% (*GAg* predictor) of BHT entries, while they constitute 49% (*GAg* predictor) to 52% (*GAs* predictor) of BHT references. The impact of kernel branches on different predictors is found to vary with the variety of mapping schemes and the degree of operating system activity.

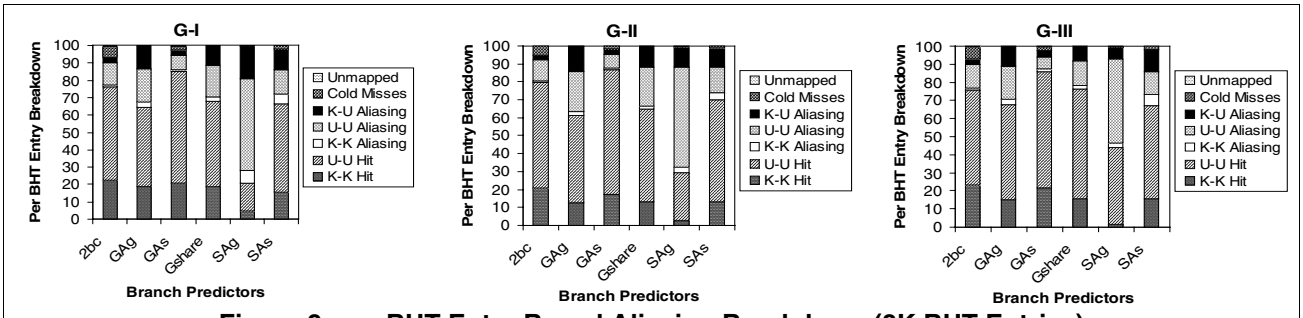

Figure 9. per BHT Entry Based Aliasing Breakdown (8K BHT Entries)

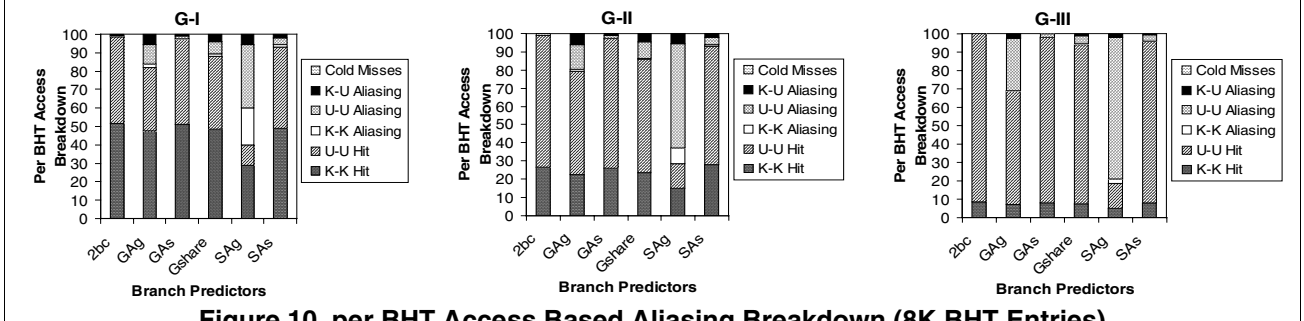

Figure 10. per BHT Access Based Aliasing Breakdown (8K BHT Entries)

Kernel/kernel aliasing (less than 10% of total aliasing in most cases) is found to be less significant than user/user aliasing (greater than 55% of total aliasing) because of the smaller number of branch sites in the kernel code. User/kernel aliasing occurs on all examined predictors. It is observed that user/kernel aliasing occurs infrequently on the *2bc* predictor since the use of address-based indexing distinguishes different branches within a large sized table. However, conflict increases in the other predictors and is more significant in global history based predictors. In *Gshare* and *GAg* predictors, for example, the user/kernel aliasing occurs in 10%-18% of BHT entries and accounts for 5%-9% of BHT references and 30%-35% of total reference aliasing. There is significant user/kernel aliasing in *SAg* and *SAs* predictors despite the local history-based BHT indexing mechanism.

Exception-driven kernel routines can be invoked at any time during program execution, which can cause arbitrary branch sequence information in the BHSR for a branch in the exception handler. This branch history ambiguity in the BHSR can further spread to user/kernel branch aliased references across the BHT counters through indexing. To demonstrate this effect, we instrumented the *Gshare* predictor simulator to capture user/kernel aliased references across the range of 8K BHT counters, shown in Figure 11. Not surprisingly, most BHT counters suffer from user/kernel aliasing, implying that the ambiguous user/kernel history information recorded in the BHSR and the randomized indexing mechanism used in *Gshare* can spread destructive user/kernel aliasing across many BHT counters.

5.2 Mispredictions Caused by User/Kernel Branch Aliasing

Aliasing may not directly imply reduced prediction accuracy since a branch that will eventually be aliased may be executed enough times to amortize the context switch cost before the conflict occurs. During the execution of Java programs, the instruction streams of user applications (e.g., JIT compiler and translated native methods) and kernel instructions (invoked by interrupts, exceptions or system calls) alternate with each other. So branches from user and kernel space alternate in trace order.

To analyze how user/kernel branch aliasing impacts prediction accuracy, simulations were run with various branch prediction schemes using 8K BHT entries and full-system, user-only (user), and kernel-only (OS) traces. For the full-system simulations, the branch prediction simulators were instrumented to attribute and normalize the aggregate misprediction rate as perceived by the user (shown as full system(user) in the figure) and by the kernel (shown as full system(OS)). The results are shown in Figure 12.

As observed earlier, for a given prediction scheme, the G-I benchmarks suffer more user/kernel aliasing than G-II and G-III benchmarks. Unsurprisingly, misprediction rates for user or kernel code alone are always better than the misprediction rates as perceived by one with the other causing aliasing. For example, on benchmark group G-I under *Gshare*, eliminating user/kernel aliasing drops the user misprediction rate from 10.5% to 6.3% and kernel misprediction rate from 3.3% to 1.1% respectively. The negative effect of aliasing on prediction accuracy is more pronounced in the two-level schemes with large history depths (e.g. *GAg* and *Gshare*) than in locally oriented schemes that rely on smaller history depths (e.g. *SAs*). Also, the *2bc* results appear similar, indicating that little user/kernel aliasing is occurring.

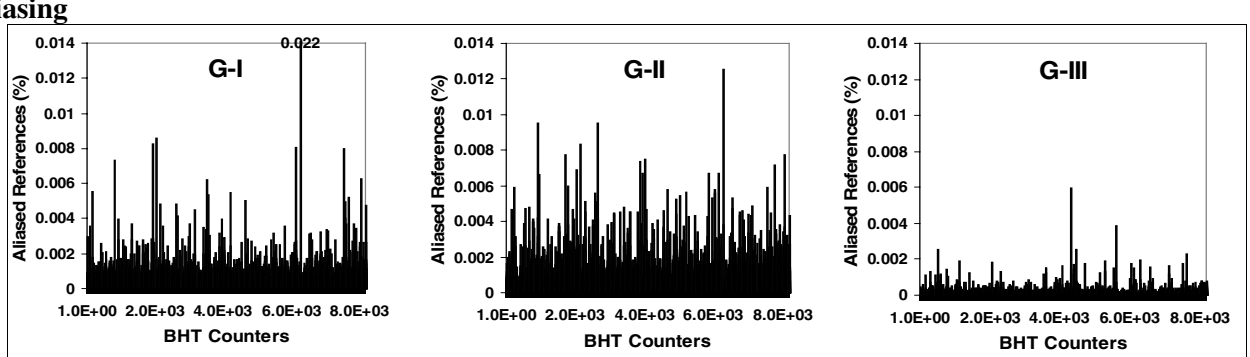

Figure 11. Distribution of User/Kernel Aliasing on Gshare BHT Counters (8K)

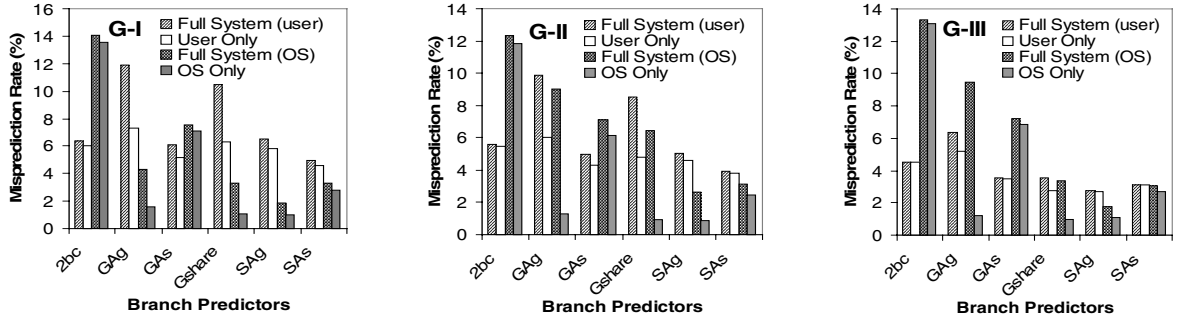

Figure 12. Impact of User/Kernel Branch Aliasing

5.3 Reducing the Impact of User/Kernel Branch Aliasing

The results in Figure 12 imply that eliminating user/kernel branch aliasing improves prediction accuracies for both user code and kernel code. A natural way to achieve this goal is to split the unified predictor resource and provide a dedicated branch predictor for each mode. Figure 12 also shows that kernel and user codes favor different branch prediction schemes. For example, with aliasing, the kernel has better branch prediction accuracy using *SAg* and *SAs* predictors. Without aliasing, *GAg* and *Gshare* predictors also yield accurate results for kernel code. This implies that splitting branch predictors and using separate schemes for the two modes can result in higher performance with reduced hardware.

Figure 13 compares the performance of unified predictors, predictors with hardware resources split half-and-half between user and kernel code, and 'hybridized' predictors that use split hardware and separate schemes for each of the two modes. The results are shown on the 6 studied benchmarks individually. It is found that the simple half-and-half split branch predictors using *GAg* and *Gshare* reduce mispredictions for both modes over all benchmarks. For those two predictors, splitting yields 30% (*db*) to 90% (*jack*) prediction accuracy improvement in kernel code and 9% (*compress*) to 50% (*jack*) in user code. The half-and-half split also benefits kernel branch prediction for *SAg*, *SAs* and *GAs* predictors but the percentage of improvement is smaller (less than 5%). Another observation is that the impact of splitting on kernel branches varies with different benchmarks. For example, on benchmark *jack*, the splitting contributes to misprediction reduction for all examined schemes. The simple half-and-half splitting policy is found to penalize prediction accuracy in user codes on *SAg* and *SAs* predictors by increasing the misprediction rates up to 7%.

Even better predictors can be obtained by combining hybridizing with splitting. For example, the misprediction rates of benchmark *jack* drop from 9.6% on a unified *Gshare* predictor with 4K BHT to 4.1% on a hybrid *GAs*(U) +*Gshare*(K) configuration of the same size. The *SAs*(U)+ *Gshare*(K) and *GAs*(U)+ *Gshare*(K) configurations provide the best performance on most of the studied benchmarks. Among other examined hybrid split predictors, we find that the simple *2bc*(U)+*Gshare*(K), *2bc*(U)+*GAg*(K) configurations yield performance comparable to that of the best predictors.

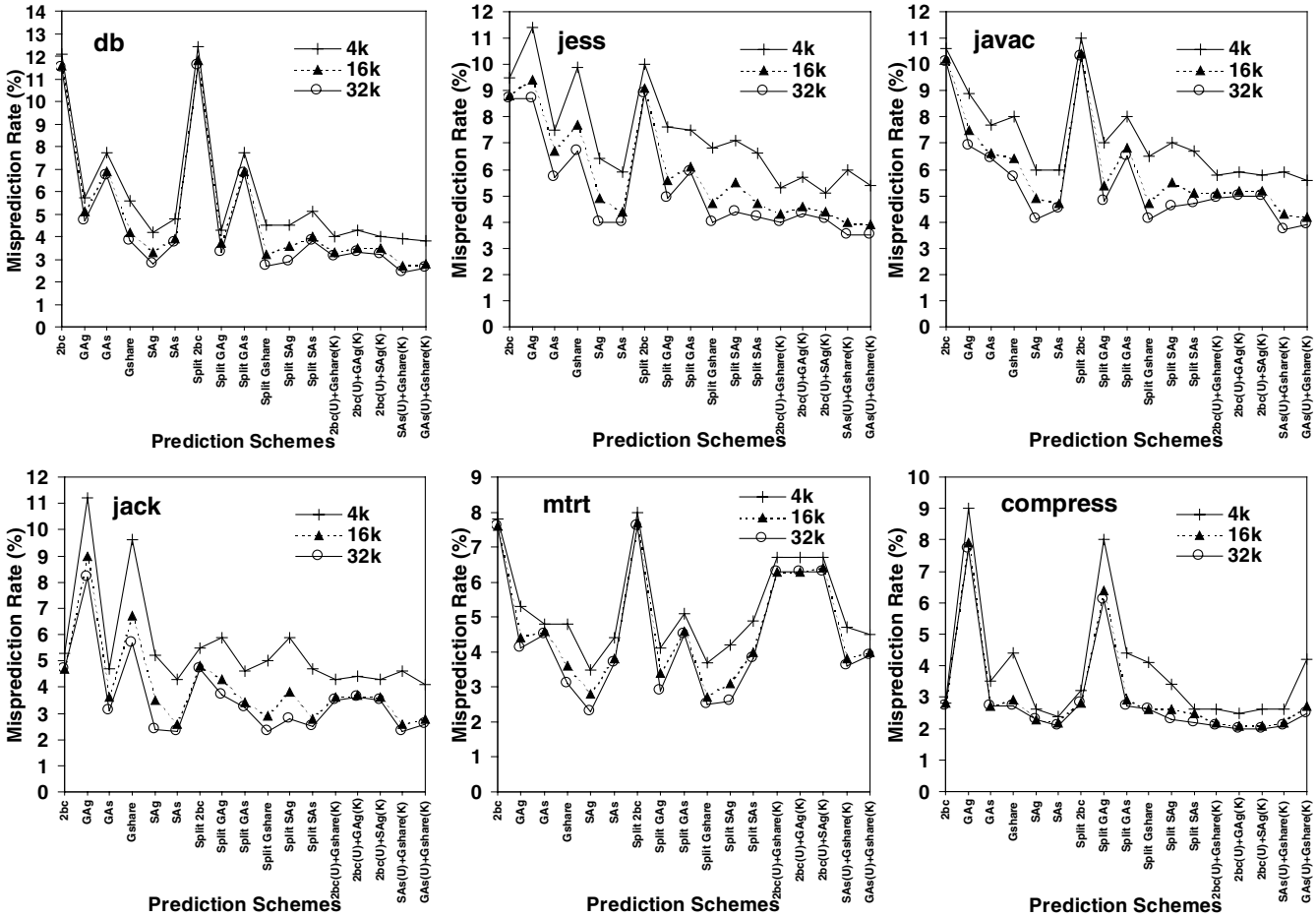

Figure 13. Performance of Half-and-Half Split and Hybridized Split Predictors

6. Conclusions

This paper presents the results of extensive simulations of a variety of state-of-the-art branch predictor schemes on a full system simulation framework running a commercial operating system and emerging Java workloads.

We find that branch predictors with the best performance for user code also have the best performance for kernel code as well as the best performance for the overall system code. However, the advanced predictors often benefit operating system code more than user code, and in some cases may not outperform simpler predictors. Also, overall system misprediction rates do not track user misprediction rates in the presence of high operating system activity, which is the case for some Java benchmarks. Complexity analysis shows that simpler prediction schemes may be more cost-effective than more complicated schemes for emerging workloads such as Java.

Kernel/kernel aliasing was found to be a smaller percentage of aliasing than user/user aliasing for all examined branch predictors. Also, user/kernel aliasing occurs in all predictors and can be significant. Simulation results indicate that prediction accuracies can be improved by using split hardware resources with different prediction schemes for user and kernel code without consuming additional hardware resources. Hybrid prediction schemes can also be used to identify attractive tradeoffs between predictor performance and complexity. As future work, we plan to model the power consumption and access latencies of hybrid split branch predictors.

References

1. T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson and M. Wolczko, Compiling Java just in time, IEEE Micro, vol. 17, pages 36-43, May 1997
2. M. Evers, P. Y. Chang and Y. N. Patt, Using Hybrid Branch Predictors to Improve Branch Prediction Accuracy in the Presence of Context Switches, In Proceedings of the 23rd Annual International Symposium on Computer Architecture, pages 3-11, 1996
3. N. Gloy, C. Young, J. B. Chen and M. D. Smith, An Analysis of Dynamic Branch Prediction Schemes on System Workloads, In Proceedings of the 23rd Annual International Symposium on Computer Architecture, pages 12-21, 1996
4. C. C. Lee, I. C. K. Chen, and T. Mudge, The Bi-Mode Branch Predictor, In Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture, pages 4 - 13, 1997
5. 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
6. T. Li and L. K. John, Understanding Control Flow Transfer and its Predictability in Java Processing, In Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software, pages 65-72, 2001
7. S. McFarling, Combining Branch Predictors, WRL Technical Note TN-36, Digital Equipment Corporation, June 1993
8. R. Nair, Dynamic Path-Based Branch Correlation, In Proceedings of the 28th Annual International Symposium on Microarchitecture, pages 15-23, 1995
9. T. Li, L. K. John, A. Sivasubramaniam, N. Vijaykrishnan and J. Rubio, Understanding and Improving Operating System Effects in Control Flow Prediction, In Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, 2002
10. J. A. Redstone, S. J. Eggers and H. M. Levy, An Analysis of Operating System Behavior on a Simultaneous Multithreaded Architecture, In Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, pages 245-256, 2000
11. 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
12. S. Sechrest, C-C. Lee, and T. Mudge, Correlation and Aliasing in Dynamic Branch Predictors, In Proceedings of the 23rd Annual International Symposium on Computer Architecture, pages 22-32, 1996
13. SPEC JVM98 Benchmarks, http://www.spec.org/osg/jvm98/
14. E. Sprangle, R. S. Chappell, M. Alsup and Y. N. Patt, The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference, In Proceedings of the 24th Annual International Symposium on Computer Architecture, pages 284-291, 1997
15. T. Y. Yeh, and Y. N. Patt, A Comparison of Dynamic Branch Predictors that Use Two Levels of Branch History, In Proceedings of the 20th Annual International Symposium on Computer Architecture, pages 257-266, 1993
16. C. Young, C. Gloy and M. D. Smith, A Comparative Analysis of Schemes for Correlated Branch Prediction, In Proceedings of the 22nd Annual International Symposium on Computer Architecture, pages 276-286, 1995