Understanding and Improving Operating System Effects in Control Flow Prediction

Tao Li[†], Lizy Kurian John[†], Anand Sivasubramaniam[*], N. Vijaykrishnan[*] and Juan Rubio[†]

[†]Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712
{tli3,ljohn,jrubio}@ece.utexas.edu

[*]Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802
{anand, vijay}@cse.psu.edu

ABSTRACT

Many modern applications result in a significant operating system (OS) component. The OS component has several implications including affecting the control flow transfer in the execution environment. This paper focuses on understanding the operating system effects on control flow transfer and prediction, and designing architectural support to alleviate the bottlenecks. We characterize the control flow transfer of several emerging applications on a commercial operating system. We find that the exception-driven, intermittent invocation of OS code and the user/OS branch history interference increase the misprediction in both user and kernel code.

We propose two simple OS-aware control flow prediction techniques to alleviate the destructive impact of user/OS branch interference. The first one consists of capturing separate branch correlation information for user and kernel code. The second one involves using separate branch prediction tables for user and kernel code. We study the improvement contributed by the OS-aware prediction to various branch predictors ranging from simple Gshare to more elegant Agree, Multi-Hybrid and Bi-Mode predictors. On 32K entries predictors, incorporating OS-aware techniques yields up to 34%, 23%, 27% and 9% prediction accuracy improvement in Gshare, Multi-Hybrid, Agree and Bi-Mode predictors, resulting in up to 8% execution speedup.

1. INTRODUCTION

Every once in a while in systems research, we come to the point where we need to evaluate how well is the hardware suited for a given application, how well is it accommodating the operating system (OS), and how well is the application exploiting an OS's capabilities. Such issues are extremely important in order to fine tune system performance since we find that the three subsystems - application (workload), OS and hardware - are constantly evolving, and many times quite independently [1].

The nature and diversity of workloads have seen substantial changes that we need to go back to understanding the interplay of the three subsystems based on these new workloads. In particular, we note the growing importance of the OS in emerging application environments, with OS services being invoked much more often. Several recent studies [22][20] have reported high operating system contribution to the overall execution time. In commercial applications such as databases and web services, the OS component has been observed to reach as high as 55% of the execution time in some of the workloads considered in this paper. Li et al. [11] also report higher OS involvement in Java applications compared to traditional SPEC workloads. The reason for the higher OS involvement in all these emerging environments is because the applications are in general multi-threaded and exercise the I/O subsystem much more extensively. This trend is likely to continue in the near future and it is very important to consider the operating system not only for complete system evaluations as other studies have pointed out [21], but also when attempting to optimize the hardware and/or the application [32]. A detailed characterization of the interactions between the application-OS-hardware can have considerable ramifications in the design of each system component, and this paper takes a step in that direction.

We focus on one specific issue that has long been considered an important issue for performance optimization of state-of-the-art processors - control flow prediction. Current high performance processors provision aggressive support for Instruction Level Parallelism (ILP) and have deep pipelines to keep cycle times low. The delivered ILP and pipelining performance is critically dependent on being able to accurately predict the control (branch) flow in the program, so that we can execute more useful instructions and avoid stalling/squashing the pipeline.

Branch predictors for control flow prediction have been studied extensively with different programs [29][31][23][15] and also with OS effects [8]. The OS affects control flow predictability by introducing the additional user/OS branch aliasing in branch predictor tables. The negative impact of kernel branches on branch prediction has been reported in [8]. We also find that kernel code nearly doubles the misprediction rates in 7 out of 13 of our benchmarks in a Gshare predictor (Figure 1).

Branch aliasing characterization shows that user/OS aliasing contributes to up to 24% of all misprediction and 46% of aliasing misprediction in the benchmarks studied in this paper. While eliminating all branch prediction aliasing is not trivial, it is our belief that the destructive user/OS part can be alleviated with appropriate architectural support. There are numerous branch predictors that have been proposed to address different situations [30][14][7][25][10][4][15][6]. These prediction mechanisms have paid less attention to the OS requirements and no particular scheme was proposed on tuning control flow prediction hardware for the OS. Our intention in this paper, however, is not to propose a new predictor to add to this list. Rather, it is to understand the fundamental nature of the user/OS interference and suggest simple and cost-effective optimizations that one can incorporate into any predictor to alleviate the impact of the OS activity on the control flow prediction.

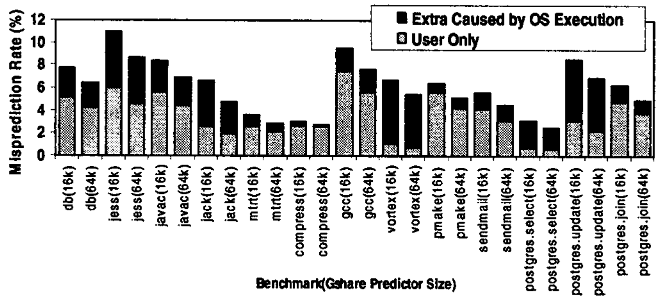


Figure 1. Impact of User/OS Execution on Branch Prediction

Adhering to this philosophy, we investigate what causes the execution of a spectrum of applications with significant OS involvement to give worse branch prediction in the user and kernel modes by characterizing their execution using complete system simulation. This investigation shows that the interference between the branches in the user and kernel modes is leading to this poor performance. User and kernel branches have different characteristics (such as the direction bias) that cause the history information used by the predictors - and shared by both the user and kernel - to become polluted. Such pollution would not have happened if we had a separate predictor for each mode.

These observations lead us to advocate separating out branch prediction logic for user and kernel modes. By doing this, we are reducing the interference between the two. This approach can be easily integrated into existing prediction schemes without significantly complicating the logic. Separating resources for OS to reduce the user/OS interference exists for other resources (e.g., TLB, memory etc.). However, to our knowledge, this paper presents the first study and quantitative analysis on orchestrating control flow prediction resources for OS.

The rest of this paper is organized as follows. In section 2, we characterize OS branch behavior in different applications. We also quantify the effect of user/OS branch aliasing or interference[1]. Section 3 introduces OS-aware prediction designed specifically to reduce user/OS branch aliasing and evaluates the improvement contributed by the OS-aware techniques to various branch prediction strategies. Finally, conclusions are provided in Section 4.

[1] We use the terms branch aliasing and branch interference interchangeably in this paper.

2. CHARACTERIZING OS BRANCH BEHAVIOR

In this section, we use simulation of complete system activity to characterize OS branch execution and evaluate its impact on branch predictability. We use the SimOS [22] full system simulation environment and select thirteen benchmarks with significant OS involvement. We provide a brief overview of the selected applications. A set of six Java applications from the SPECjvm98 [24] suite (s1 dataset) is executed using a commercial JDK [13] from Sun Microsystems simulated on top of IRIX 5.3. *Vortex* is a database manipulation code and *gcc* is a compiler code from the SPECint95 benchmark suite. *Pmake* is a program development workload that is a variant of the compiled phase of the modified Andrew benchmark employed in [21]. The *sendmail* benchmark forwards email messages to local accounts on the system using the Simple Mail Transport Protocol (SMTP). The sizes of the messages vary from 1K to 1.5M. We also use three benchmarks that use PostgreSQL [19][26], a relational database management system (DBMS). The database is populated with relational tables for the TPC-C benchmark [27]. We evaluate the execution of three specific queries on this data set: a sequential table scan of a table with 1 million rows and a selectivity of 3% (*postgres.select*), an update to a field of a 300,000 row table (*postgres.update*) and a nested loop join involving two tables of sizes 11MB and 24KB (*postgres.join*). The SPECjvm98 applications, *pmake* and *gcc* are all executed to completion. All other applications are simulated for billion-instruction execution. Table 1 summarizes the complete system branch execution statistics of each studied benchmark.

As illustrated in Table 1, the OS activity in the selected benchmarks ranges from 6% in *compress* to as high as 55% in *postgres.update*. The kernel portion of dynamic branch instances can be found to constitute a significant part in these applications. On the average, kernel branches, which include loops, error/bound checking, and other routine conditionals, constitute 27% of branch sites and 30% of dynamic branch instances in our benchmark executions. Branches have been found to be more frequent in OS (than in user mode) [20] because it has to be designed to handle all possible situations (i.e., abundant error and bound checking). Further, the OS functions are performed not just for one process/application but also for the system as a whole (other daemons, periodic book-keeping duties etc.).

2.1 Context Switch Profile and Branch Distribution

During the execution, branch instructions from user and OS code get interspersed. OS is activated either voluntarily by a system call from the application, or from a call by some other application, or implicitly by some underlying periodic/asynchronous (timer/device interrupt) mechanism. The inter-mingling of user and kernel branches can affect their behavior, compared to the execution when they were isolated from each other. Figure 2 shows the average number of executed branches in each mode per context invocation on the studied benchmarks. In all benchmarks except *db* and *postgres.update*, OS exercises fewer branches than user code in each visit to that mode. For benchmarks *db* and *postgres.update*, OS service *read* and *write*, which consists of far more branch instructions, dominates OS execution, causing higher average number of executed branches in OS.

Table 1. Complete System Branch Execution Statistics

Benchmarks	Total Instructions (M)	% OS Execution	# of Context Switches between User/OS	Conditional Branch Statistics			
				User		OS	
				Static Sites (percentage)	Dynamic Instances (percentage)	Static Sites (percentage)	Dynamic Instances (percentage)
db[1]	201	31	0.9M	33,957 (85%)	13.1M (40%)	6,016 (15%)	19.7M (60%)
jess[2]	467	30	4.9M	38,654 (86%)	36.0M (56%)	6,037 (14%)	28.3M (44%)
javac[3]	366	19	2.0M	38,815 (86%)	34.8M (63%)	6,070 (14%)	20.8M (37%)
jack[4]	1,782	17	23.5M	40,640 (87%)	210.7M (84%)	6,142 (13%)	40.5M (16%)
mtrt[5]	1,431	7	5.9M	36,629 (86%)	195.7M (89%)	6,099 (14%)	23.3M (11%)
compress[6]	2,428	6	11.8M	33,907 (85%)	406.4M (94%)	6,081 (15%)	26.1M (6%)
gcc[7]	1,036	8	5.0M	13,570 (74%)	138.9M (91%)	4,696 (26%)	13.8M (9%)
vortex[8]	1,811	8	21.5M	4,108 (78%)	133.5M (92%)	1,189 (22%)	12.0M (8%)
pmake[9]	1,117	17	1.0M	11,651 (69%)	122.5M (78%)	5,273 (31%)	33.8M (22%)
sendmail[10]	1,494	54	1.4M	4,516 (45%)	139.3M (65%)	5,553 (55%)	75.1M (35%)
postgres.select[11]	1,516	38	5.6M	8,417 (58%)	107.2M (53%)	6,201 (42%)	93.6M (47%)
postgres.update[12]	1,438	55	6.4M	8,144 (56%)	83.4M (36%)	6,325 (44%)	149.1M (64%)
postgres.join[13]	1,849	15	5.9M	8,606 (59%)	220.7M (75%)	6,099 (41%)	72.7M (25%)

1.Performs multiple database functions on a memory resident database
2.Java expert shell system based on NASA's CLIPS expert system
3.The JDK 1.0.2 Java compiler compiling 225,000 lines of code
4.Parser generator with lexical analysis, early version of what is now JavaCC
5.Dual-threaded raytracer
6.Modified Lempel-Ziv method (LZW) to compress and decompress large file
7.Compiles pre-processed source into optimized SPARC assembly code
8.A full object oriented database
9.Two parallel compilation processes compile the modified Andrew Benchmark[16]
10.UNIX electronic mail transport agent
11.Object-relational DBMS PostgreSQL executes a select query
12.Object-relational DBMS PostgreSQL executes an update query
13.Object-relational DBMS PostgreSQL executes a join query

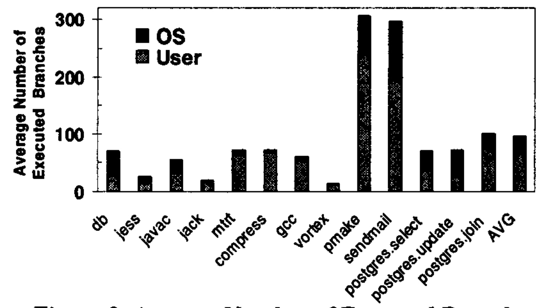


Figure 2. Average Number of Executed Branches per Visit in User and Kernel Modes

We tracked the distribution of the number of executed branches for each context switch. The profiling results for a 5,000 context switch sample of benchmark *jack* are shown in Figure 3 for user and kernel code separately. Comparing Figure 3a and 3b, one can see that the user contexts can execute far more branches than the OS contexts do. Further analysis indicates that most of these OS contexts are caused by exception driven OS routines (e.g. TLB miss and page fault) that execute very few branches. The distributions in Figure 3 for the kernel are a cause for concern since it indicates the possibility that the branch history may be not accurate for correct predictions (with interference from user mode branches). On the other hand, the user branch distribution suggests that this problem may not be as severe for the user mode. Kernel invocations are more short lived, while user execution has reasonable time quanta to work with and build history.

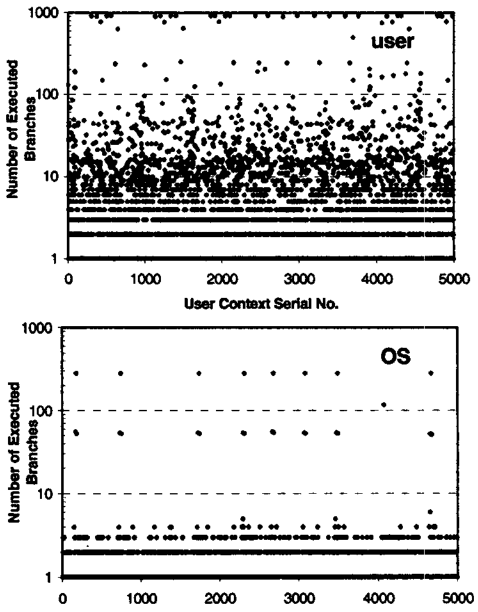


Figure 3. Executed Branches in User and OS Contexts (5,000 Sampling Contexts)

2.2 OS Branch Execution Profile

We next examine what are the dominant kernel branches, and how their performance can be affected by the user code executing between OS operations. The pie chart of Figure 4 shows the percentage of OS branches (in benchmark *jack*) executed in the different services. Additional benchmark results can be found in [12]. The top five components include: TLB miss (*TLB miss*, 39%); OS scheduling (*scheduling*, 37%); performing file and I/O services (*I/O & file system*, 5%), idle looping (*idle*, 4%); and kernel synchronization (*synchronization*, 4%).

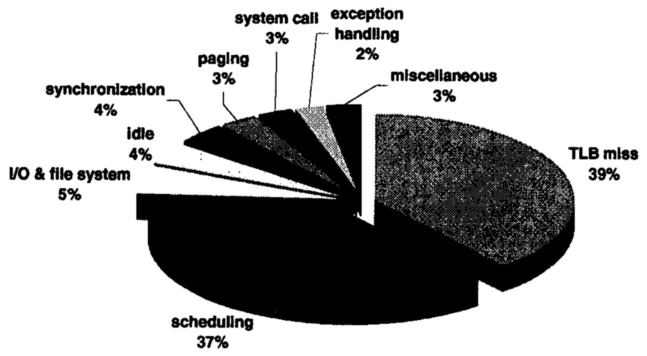


Figure 4. Where do the OS Dynamic Branches Come from?

These results show that we really need to focus mainly on the TLB handler (it is done in software on the given MIPS platform to facilitate the use of flexible page table structure and simplify the handling of sparse address spaces.) and the scheduler. Further, it should be noted that other services such as file system, synchronization etc., are directly invoked by the user code. Hence, their behavior (including that for branches) is influenced by the current state of the invoking application and the parameters of the call. So one would not like to associate the term "interference" for such services. On the other hand, TLB handling and scheduler invocations are not necessarily voluntary. It is useful to understand how the branches in these OS subsystems are invoked and whether history would have any bearing on their behavior for predictability – so that we can better understand if the predictability of these branches would be affected by the user code getting in-between.

Table 2 further shows the OS routine based branch distribution. The *utlb* is the OS TLB miss handler. The *checkRunq* routine performs scheduling (picking the next process to run). The *idle* does idle looping. Explicit system calls from user code are handled by *syscall*. The *io_splock* routine manipulates I/O spin locks to ensure that all operations to a particular I/O device are synchronized. The *exception_ipl2* is the OS general exception handler. The *bcopy* is a memory copy routine used for paging and buffer copying in OS. The *mrlock* routine gets the states of locks and semaphores. Table 2 gives further evidence of the significance of the TLB handling and scheduler subsystems on the overall branches within the OS. Though *utlb* and *checkRunq* both have high dynamic branch instances, the number of actual branch sites is quite small. We briefly go over these routines below identifying the branches in these routines and their anticipated behavior qualitatively.

The *utlb* handler has only 1 branch, and the reason for its high dynamic occurrence is that this routine is invoked frequently. The *utlb* routine is invoked directly by the hardware which is the only entity that can invoke this operation. On the other hand, the scheduler *(checkRunq)* is invoked from several places. First, this operation is needed for scheduling decisions (by consulting the ready queue) whenever the time quantum expires (triggered by timer interrupt), when I/O device activity completes (there are usually priority boosts and rescheduling may be needed) and idle looping, or even voluntarily during blocking (making semaphore, I/O requests etc.) or other process state change activities (such as termination). Consequently, it is to be noted that, while *utlb* invocations are only the consequence of application behavior, the scheduler actions are invoked from all over the OS and are invoked either asynchronously (by hardware events) or voluntarily due to system load/behavior. In all, we found there are more than 23 events that can cause *checkRunq* to be invoked.

Table 2. OS Routine Branch Characterization

OS Routine	% Dynamic Branches	Active Branch Sites
utlb	38.7	1
checkRunq	34.2	6
idle	3.89	3
syscall	2.80	14
io_splock	2.38	5
exception_ipl2	2.08	6
bcopy	1.50	6
mrlock	1.17	8

2.3 Characteristics of OS Branches

We investigate specific properties of these OS branches and their architectural implications in this subsection.

2.3.1 Branch Directions and Weakly Biased Branches in OS

It is well known that branches often have biased behavior and many branches are either usually "taken" or usually "not taken". The conventional branch history table (BHT) counters exploit this behavior to predict future outcomes of that branch. However, when branches showing different biases are mapped into the same entry of the predictor table, aliased branches update BHT counters with different directions, leading to aliasing mispredictions.

We measure branch direction distribution in order to gain more insight on bias behavior of the user and OS branches. Figure 5 shows the result on benchmark *jack*. Additional benchmark results can be found in [12]. The branch sites are categorized into 100% "taken" (always-taken), 0% "taken" (always-not-taken) and groups between them. For example, the marker "70%-79%" on X-axis implies that branch sites that fall into this category have a possibility of 70% to 79% to be "taken".

Figure 5 shows that user and OS branches behave differently in terms of the bias or direction distribution. For example, 46% of dynamic branches in OS are "always taken" while their counterparts in user code are only 15%. On the other hand, 18% of dynamic branches in OS are "always not taken" and that number in user mode can be as high as 42%. This implies that even when the strongly biased user and OS kernel branches are mapped into the same BHT counter, it is likely that they will lead to aliasing misprediction.

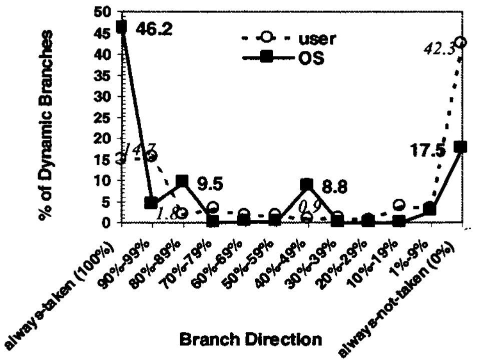


Figure 5. User and OS Branch Directions

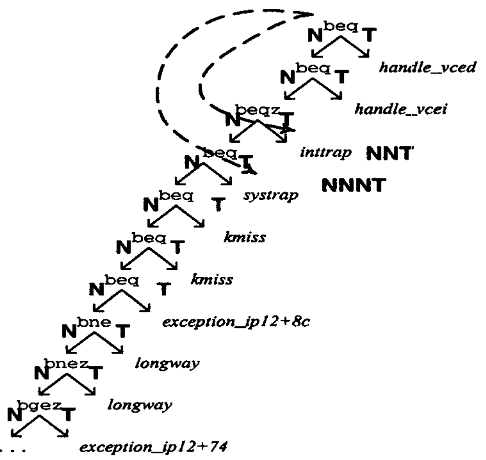
Another interesting observation in Figure 5 is that while the dominant portion of branch sites is strongly biased (i.e. always taken or always not taken) in user code, a significant number of branches are weakly biased in OS code. More precisely, we observed that 8.8% of dynamic branches that contribute to the weakly biased (with the category of 40%-49%) branches shown in Figure 5, come from a wide range of 22 kernel service routines. The weakly biased OS branches showing interleaved directions are also found on other benchmarks [12]. Among these is the *checkRunq* routine that is frequently invoked. This routine checks through queues to find out if a rescheduling decision needs to be made. Intuitively, it can be hypothesized that the execution characteristics of such a routine are more a function of the load on the system more than anything else. Even when the load does not change very much during the course of this execution, there are bursts of I/O, synchronization activity and other events that can exercise the *checkRunq* differently, causing its branch to vary direction. Weakly biased branches can be a problem to many branch predictors, which rely on the persistent history and saturated 2-bit counters for accurate branch prediction.

2.3.2 How Correlated are Kernel Branches?

We observe that many OS branches are very correlated and hence benefit from two-level predictors that exploit global history correlation. It should be noted that the *utlb* routine has a single branch that is nearly always taken. While static predictors would suffice for this branch, previous history is also a very good indicator for this particular branch that accounts for a large portion of the kernel's dynamic branches. Further, OS exception handlers frequently use binary decision trees to classify and dispatch vectored interrupts from the trap entry point to the specific fault handler. Figure 6a shows an example use of such a structure in the general exception handler (*exception_ipl2*) OS code. This handler dispatches an exception to the corresponding kernel processing routine based on the value of the exception vector. The binary decision tree based branch sequence of this handler is given in Figure 6b. It can be observed that the branches in the OS routine *inttrap* will be correlated with a NNT branching sequence while the branches in *systrap* will be correlated with a NNNT branching sequence. Hence Gshare [14] and GAg [29] predictors work extremely well with these branches.

```
0x80007dd4: <exception_ipl2>
andi $k0,$k0,0x7c
li $k1,124
beq $k0,$k1,0x80007d0c <handle_vced>
li $k1,56
beq $k0,$k1,0x80007cec <handle_vcei>
li $k1,32
beqz $k0,0x800080f0 <inttrap>
sw $at,-24524($zero)
beq $k0,$k1,0x80008770 <systrap>
li $at,8
beq $k0,$at,0x80007e78 <kmiss>
li $at,12
beq $k0,$at,0x80007e78 <kmiss>
li $at,92
beq $k0,$at,0x80007e60 <exception_ipl2+8c>
li $at,36
bne $k0,$at,0x80008274 <longway>
mfc0 $k0,$12
andi $k0,$k0,0x18
bnez $k0,0x80008274 <longway>
mfc0 $k0,$13
bgez $k0,0x80007e48 <exception_ipl2+74>
. . .
jr $at
```

(a) OS Assembly Code to Perform General Exception Handling



(b) Binary Decision Tree based Branching Sequence Corresponding to Code Shown in (a)

Figure 6. Branch Correlation in OS Code

2.3.3 Impact of Intermittent Kernel Execution on Strongly Biased Kernel Branches

Even strongly biased OS branches can experience mispredictions due to the user code interference. An example for this can be obtained from the *utlb* routine from the OS. Since the *utlb* handler needs to be very efficient, this code is usually written in assembly and is hand-optimized. There are exactly 13 instructions in this routine, with the bulk of the instructions used to read the page table entry from the memory system and load it into the TLB. There is exactly 1 branch within this code that is strongly taken. But intervening user code interference can result in mispredictions in even such strongly biased branches. Consider a correlation based branch predictor, and two scenarios of branch history shift register (BHSR) contents in Figure 7. In the absence of user code intervention, the correlation shift register may look like (a), and leads to correct prediction, whereas the intervening user code may result in the correlation information to look like (b) and result in aliasing misprediction.

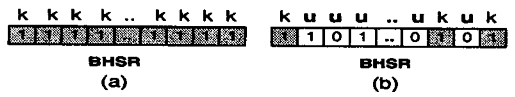


Figure 7. Impact of User/Kernel Inference on Strongly Biased Kernel Branches

2.3.4 Characterization of User/OS Aliasing

It is well known that branch aliasing, namely, several branches mapping to the same entry in the prediction tables, impacts the branch prediction accuracy. Although some of the aliasing can be neutral or constructive, a large part of the aliasing is often destructive. We performed a branch aliasing characterization to understand the impact of user/OS aliasing. In order to do that, we instrumented the branch predictor to track the mapping between branch instructions and the BHT entries. Branch aliasing is recorded whenever the branch instruction being mapped to a given BHT entry is different from what is already present at that entry. Branch aliasing is attributed to user (User/User Aliasing), kernel (OS/OS Aliasing) and the interaction between them (User/OS Aliasing). The percentages of misprediction and correct prediction caused by different aliasing categories are shown in Table 3.

Table 3. Characterization of Branch Aliasing (8K BHT Eentries Gshare, MR: Misprediction Rate)

Benchmarks	Metric	OS/OS Aliasing	User/User Aliasing	User/OS Aliasing
db (MR=4.8%)	% of Misprediction	6.2	28.2	19.4
	% of Correct Prediction	1.4	2.7	2.1
jess (MR=8.8%)	% of Misprediction	3.3	37.3	20.1
	% of Correct Prediction	1.4	6.5	3.9
javac (MR=7.1%)	% of Misprediction	3.1	34.7	16.4
	% of Correct Prediction	0.7	5.2	2.4
jack (MR=8%)	% of Misprediction	1.3	35.7	18.8
	% of Correct Prediction	0.6	7.9	4.7
mtrt (MR=4%)	% of Misprediction	1.3	23.5	10.2
	% of Correct Prediction	0.2	3.8	1.1
compress (MR=3.1%)	% of Misprediction	0.7	12.0	2.5
	% of Correct Prediction	0.1	4.7	0.2
gcc (MR=10.2%)	% of Misprediction	0.3	41.5	6.2
	% of Correct Prediction	0.1	10.5	1.9
vortex (MR=7.8%)	% of Misprediction	0.1	39.4	11.7
	% of Correct Prediction	0.0	11.8	3.8
pmake (MR=6.6%)	% of Misprediction	3.6	25.1	9.4
	% of Correct Prediction	0.5	4.6	1.0
sendmail (MR=9.3%)	% of Misprediction	22.2	9.0	23.7
	% of Correct Prediction	3.8	1.7	2.9
postgres.select (MR=3.1%)	% of Misprediction	7.4	16.0	19.7
	% of Correct Prediction	0.9	2.4	2.2
postgres.update (MR=5.7%)	% of Misprediction	7.8	18.4	22.4
	% of Correct Prediction	1.7	3.3	3.8
postgres.join (MR=5.6%)	% of Misprediction	1.1	15.0	4.5
	% of Correct Prediction	0.2	5.3	1.1

In experiments with a Gshare predictor of size 8K BHT entries, user/OS aliasing on the average contributes to the 14.2% and 2.5% of misprediction and correct prediction respectively, implying most of the user/OS aliasing are negative. The percentage of misprediction caused by user/OS aliasing does not change significantly when the predictor size is increased from 8K entries to 64K entries. This indicates that just increasing the capacity of the branch predictor will not effectively solve the user/OS aliasing problem. The user/user aliasing that many previous studies have evaluated is still important as the results observed from Table 3 indicate. However, user/OS aliasing is also a big source for mispredictions. Table 4 characterizes the impact of branch aliasing on misprediction in user and OS component. With an 8K BHT entries Gshare, approximately 22-62% of mispredictions in OS code are found to be from user/OS aliasing, suggesting that it is essential to protect kernel branch predictors from interference from user code.

Table 4. Characterization of Misprediction due to Branch Aliasing in User and OS Component (8K BHT Entries Gshare, MR: Misprediction Rate)

Benchmarks		OS/OS Aliasing	User/User Aliasing	User/OS Aliasing	MR%
db	User	--	39.0	13.5	8.6
	OS	22.3	--	34.9	2.3
jess	User	--	47.3	12.8	12.3
	OS	15.5	--	47.7	4.3
javac	User	--	42.0	10.0	9.3
	OS	17.9	--	47.0	3.5
jack	User	--	43.9	11.6	7.8
	OS	6.9	--	50.4	9.4
mtrt	User	--	26.6	5.8	3.9
	OS	11.5	--	44.0	4.7
compress	User	--	12.5	1.3	3.1
	OS	16.8	--	32.0	2.1
gcc	User	--	43.6	3.3	10.6
	OS	6.7	--	62.0	5.8
vortex	User	--	44.7	6.6	7.5
	OS	1.0	--	49.5	11.3
pmake	User	--	28.8	5.4	7.2
	OS	28.0	--	36.2	4.3
sendmail	User	--	19.9	26.2	6.3
	OS	40.5	--	21.6	14.9
postgres.select	User	--	26.7	16.5	3.5
	OS	18.4	--	24.5	2.6
postgres.update	User	--	29.3	17.9	9.6
	OS	21.0	--	29.9	3.5
postgres.join	User	--	16.1	2.4	7.0
	OS	16.2	--	33.5	1.6

In summary, we observe that user/OS branch aliasing can significantly deteriorate branch prediction accuracy. This is primarily attributed to the exception-driven and intermittent kernel branch execution that causes inaccurate branch history information in BHSR. Moreover, user and kernel branches have different bias distribution, which in turn spreads user-kernel branch aliasing references across a wide range of BHT entries. The above observations motivate the need for OS-aware branch prediction techniques.

3. ALLEVIATING IMPACT OF USER/OS INTERFERENCE

It is clear from the prior sections that user and kernel code possess different branch behavior, often resulting in conflicts in unified structures that capture branch history. In subsections 3.1 and 3.2, we present two structures that aim to alleviate the destructive impact of OS branch execution on branch predictability.

During the initial period of a context switch, both user and kernel history patterns coexist in history capturing structures. In Gshare and any correlation based predictor, this can happen in shift registers (BHSRs) that capture correlation between branches and/or branch history tables (BHTs). One solution is to use separate shift registers to individually keep track of branch correlation and another solution is to use separate BHTs.

3.1 Split BHSR Predictor

We illustrate our OS-aware techniques in the context of a Gshare predictor, but it can be applied to other correlation-based predictors as well. A Gshare predictor with split correlation history shift registers (i.e. split BHSR predictor) is illustrated in Figure 8. The split BHSR predictor functions exactly the same as a conventional Gshare predictor except that two dedicated BHSRs (i.e., U-BHSR for user and K-BHSR for kernel) are used to gather branch correlation patterns and to generate BHT indexing. By using K-BHSR for kernel branches, the split BHSR predictor overcomes the loss of branch history patterns in kernel mode. Meanwhile, the split BHSR predictor dynamically switches between BHSRs when a context switch occurs, preventing the BHT indexing ambiguity during the initial stages of a context switch.

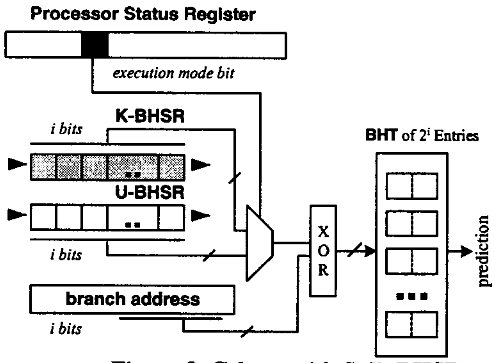


Figure 8. Gshare with Split BHSR

3.2 Split Predictor

The proposed split BHSR predictor aims to preserve accurate BHT counter indexing during a context switch. However, user/OS aliasing can still occur when user and kernel branches have the same XORed global history pattern, but opposite biases. Due to their different branch bias distribution, user and kernel branches can update BHT counters in different manners. To reduce the destructive user/OS branch aliasing in BHT, we propose the use of split BHT for user and kernel code, which yields split predictor, as shown in Figure 9. This predictor eliminates the destructive user/OS aliasing by using separate correlation and history information for user mode and kernel mode. It is also observed that when branch history tables are split into user and kernel parts, the kernel BHT can be smaller than the user BHT because of the fewer active branch sites in kernel (as shown in Table 1). Due to the difficulty in creating a 7:1 or 3:1 split (due to the user BHT becoming not power of 2), we kept the user BHT at half the size of the original Gshare and allocate kernel BHT with a fixed size of 2K entries in our experiments.

Separating out kernel branches can easily be done at run time by using the Processor Status Register (PSR). Typically, in a microprocessor a set of PSR bits is used to record and identify kernel-user execution mode or privilege level. For example, MIPS R10000 [28] uses KSU field in PSR to identify current execution mode and Intel's next generation IA-64 Itanium (Merced) [5] uses PSR.cpl to determine one of 4 privilege levels (level 0-3). The corresponding field in PSR can be used to select the appropriate predictor. At runtime, instructions from a fetch unit are filtered into an active part of prediction resource (user or kernel, depending on execution mode).

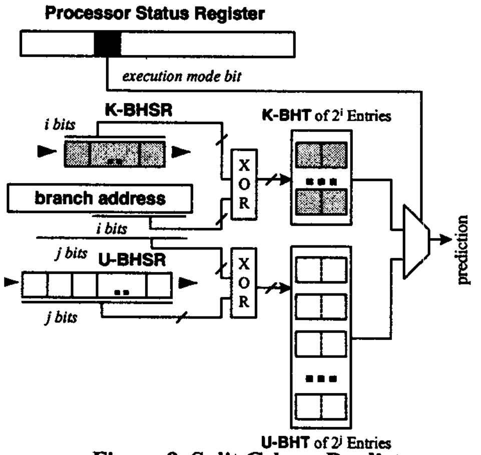


Figure 9. Split Gshare Predictor

In summary, the split BHSR predictor and split predictor are designed specifically to reduce user/OS branch aliasing without adding extra hardware for branch de-aliasing. They consume equivalent or less resource than a conventional predictor.

3.3 Integrating OS-aware Prediction Techniques with other Predictors

Splitting user and kernel prediction resources is a technique suggested by our characterization study, not necessarily a particular predictor. We surveyed literature to identify branch predictors, which may be poised to handle branches with the characteristics unveiled in the earlier sections. Although not targeted for user/OS branch interference, Multi-Hybrid [7], Agree [25] and Bi-Mode [10] schemes do contain mechanisms tailored for branches with heterogeneous characteristics and/or de-aliasing. Table 5 summarizes these schemes, and the additional cost used for branch de-aliasing. The sizes of all the predictors are normalized to Gshare to give an indication of the associated hardware cost.

All these predictors contain a Gshare predictor or a Gshare indexing [7][25][10]. To integrate the proposed techniques, we simply replace the conventional Gshare component used in the above predictors with the proposed OS-aware split-BHSR Gshare predictor and split Gshare predictor.

Table 6a shows the average (of the 13 studied benchmarks) misprediction rates of each baseline predictor and the percentage of misprediction reduction by incorporating the OS-aware techniques proposed in this paper. Table 6b further illustrates the breakdown of the misprediction reduction in user and OS parts, for each individual benchmark.

As described in subsection 3.1, split BHSR predictor only separates the branch history shift registers. The partitioning of the

BHT for user or OS happens dynamically. The resource available for the code is not less than that in the baseline. Hence, split BHSR predictor is never inferior to the baseline. Split predictor is at times worse than the baseline. In split predictor, the partitioning of the BHT between user and kernel code is done statically. Both the user and kernel BHTs are smaller than the unified BHT in the baseline configuration. In the configurations studied in this paper, the U-BHT is only 50% of the baseline BHT, and the K-BHT is fixed at 2K entries in all cases. Hence, the overall size of the split predictor BHT is not much greater than 50% of the BHT in the baseline. A 2K K-BHT is seen to be sufficient to capture all history patterns in the OS code and except in *postgres.update*, the mispredictions in OS code goes down. For the user part, the small size of the U-BHT (4K BHT entries) can detrimentally affect the performance on benchmarks *compress*, *gcc*, *pmake*, *postgres.select* and *postgres.join*.

On the average, with a 32K BHT entries Gshare, incorporating OS-aware split BHSR predictor and split predictor reduces 34% and 22% of the misprediction. OS-aware predictions also reduce the misprediction of Multi-Hybrid, Agree and Bi-Mode predictors. For instance, compared with the 32K BHT entries baseline predictors, OS-aware Multi-Hybrid, Agree and Bi-Mode predictors yield up to 23%, 27% and 9% prediction accuracy improvement respectively, implying that OS-aware predictions still provide significant improvements on some of the most powerful predictors.

As shown in Table 6a and Table 6b, split BHSR predictor outperforms split predictor on most of the de-aliasing predictors we examined. Considering overall performance, in more than half the cases, the performance gain due to the elimination of user/OS aliasing by split predictor outweighs the performance loss due to individually using smaller prediction tables for each part. More precisely, for example, the OS-aware split predictor reduces 22% of misprediction on a conventional Agree predictor of 32K BHT entries, using only 18K entries BHT consisting of a 16K entries U-BHT and a 2K entries K-BHT.

Table 5. A Comparison of Several Branch De-aliasing Schemes

Predictor	Description of Feature to Exploit Heterogeneous Branches or De-aliasing	Additional Branch De-aliasing Hardware	Predictor Size Normalized to Gshare (8k-256k)
Gshare [14]	Consists of one correlation shift register (BHSR) and one BHT. BHSR is XORed with branch address bits of a branch address to index BHT entries. The XORing helps to reduce aliasing effects.	0	1
Multi-Hybrid [1, 2] [7]	Consists of multiple single-scheme components: simple 2-bit (2bc), GAs, Gshare, Pshare and always taken predictor. Use of simple 2-bit predictors (2bc) and static predictors as components of the Multi-Hybrid predictor provides quick warm up after a context switch.	5×2K predictor selection counters in BTB	1.04-2.25
Agree [25]	Converts instances of destructive aliasing into either constructive or neutral aliasing by attaching each branch with a biasing bit that predicts the most likely outcome of that branch.	2K biasing bits in BTB	1-1.13
Bi-Mode [10]	Uses separate history tables for taken and not-taken branches, and a selection branch history table. This classification helps to alleviate destructive aliasing while keeping the harmless aliasing together.	the third BHT for dynamic bias selection	1.5
OS-aware split BHSR predictor [this paper]	OS-aware Gshare predictor uses separate shift registers (U-BHSR and K-BHSR) for capturing path history patterns.	1 shift register	1
OS-aware split predictor [this paper]	OS-aware Gshare predictor that uses separate branch history tables for user and kernels. Kernel BHT is 2K and User BHT is 50% of Gshare.	consumes less BHT resource than Gshare	0.51-1

1. Our simulated Multi-Hybrid does not include AVG predictor [3] because it needs source recompilation which often is difficult for commercial and complicated software like OS and JVM.

2. As indicated by [7], we allocate half of the total budget for Gshare, a quarter of the total budget for Pshare, and 1/8 for 2bc and GAs respectively. The priority ordering of the component predictors is 2bc, GAs, Gshare, Pshare and always taken scheme.

Table 6a. Misprediction Reduction by Introducing OS-aware Prediction

Schemes	Metric	Size (Number of BHT entries, not including de-aliasing overhead)					
		8k	16k	32k	64k	128k	256k
Gshare	Misprediction(in %)	14.03	12.35	10.89	9.64	8.66	8.00
Gshare+OS-aware Split BHSR Predictor	% of Misprediction Reduction	31%	33%	34%	32%	31%	29%
Gshare+OS-aware Split Predictor	% of Misprediction Reduction	20%	24%	22%	20%	17%	15%
Multi-Hybrid	Misprediction(in %)	10.87	9.53	8.58	7.66	6.96	6.30
Multi-Hybrid+OS-aware Split BHSR Predictor	% of Misprediction Reduction	21%	22%	23%	23%	22%	22%
Multi-Hybrid+OS-aware Split Predictor	% of Misprediction Reduction	13%	12%	13%	11%	10%	8%
Agree	Misprediction(in %)	12.59	11.41	10.46	9.66	9.13	8.78
Agree+OS-aware Split BHSR Predictor	% of Misprediction Reduction	27%	27%	27%	26%	25%	24%
Agree+OS-aware Split Predictor	% of Misprediction Reduction	19%	22%	22%	20%	20%	19%
Bi-Mode	Misprediction(in %)	7.70	6.95	6.42	6.07	5.79	5.57
Bi-Mode+OS-aware Split BHSR Predictor	% of Misprediction Reduction	10%	9%	9%	9%	9%	9%
Bi-Mode+OS-aware Split Predictor	% of Misprediction Reduction	4%	2%	1%	1%	0%	0%

Table 6b. Breakdown of Misprediction Reduction by Introducing OS-aware Prediction (8K BHT Entries)

Benchmarks		Schemes and % of Misprediction Reduction							
		Gshare + OS-aware		Multi-Hybrid + OS-aware		Agree +OS-aware		Bi-Mode + OS-aware	
		Split BHSR Predictor	Split Predictor	Split BHSR Predictor	Split Predictor	Split BHSR Predictor	Split Predictor	Split BHSR Predictor	Split Predictor
db	User	28%	23%	20%	15%	21%	17%	9%	8%
	OS	28%	8%	7%	11%	15%	7%	7%	10%
	Full-System	28%	19%	16%	14%	20%	16%	8%	8%
jess	User	39%	31%	31%	25%	34%	27%	13%	8%
	OS	52%	42%	12%	15%	44%	36%	13%	20%
	Full-System	42%	34%	28%	23%	36%	29%	13%	10%
javac	User	28%	19%	20%	13%	24%	17%	8%	4%
	OS	40%	36%	10%	20%	42%	41%	9%	18%
	Full-System	30%	22%	18%	14%	27%	21%	8%	6%
jack	User	57%	47%	47%	39%	51%	42%	21%	13%
	OS	79%	82%	29%	49%	64%	70%	43%	53%
	Full-System	61%	53%	46%	40%	53%	46%	23%	17%
mtrt	User	27%	15%	27%	19%	20%	11%	7%	4%
	OS	60%	59%	15%	23%	49%	48%	19%	27%
	Full-System	31%	20%	25%	19%	22%	15%	8%	6%
compress	User	11%	-27%	10%	-3%	7%	-30%	3%	2%
	OS	43%	29%	7%	11%	19%	12%	8%	13%
	Full-System	12%	-25%	10%	1%	7%	-29%	3%	3%
gcc	User	16%	2%	10%	-1%	12%	2%	10%	-1%
	OS	46%	55%	3%	26%	62%	68%	14%	31%
	Full-System	18%	5%	10%	0%	15%	7%	10%	1%
vortex	User	76%	63%	71%	48%	73%	65%	35%	28%
	OS	96%	97%	30%	54%	98%	99%	67%	77%
	Full-System	78%	68%	70%	48%	78%	72%	37%	31%
pmake	User	8%	-6%	4%	-11%	6%	-7%	4%	-6%
	OS	11%	2%	2%	8%	7%	13%	3%	8%
	Full-System	8%	-4%	4%	-8%	6%	-5%	4%	-4%
sendmail	User	5%	3%	1%	0%	3%	2%	2%	1%
	OS	5%	0%	3%	1%	3%	2%	2%	2%
	Full-System	5%	1%	2%	0%	3%	2%	2%	2%
postgres.select	User	56%	45%	47%	12%	50%	48%	36%	-34%
	OS	27%	8%	17%	22%	26%	29%	14%	13%
	Full-System	45%	30%	35%	16%	40%	40%	26%	-14%
postgres.update	User	35%	30%	25%	24%	25%	25%	23%	21%
	OS	14%	-10%	6%	6%	9%	17%	5%	5%
	Full-System	27%	14%	17%	17%	19%	22%	16%	15%
postgres.join	User	12%	-6%	8%	-1%	10%	-6%	3%	-6%
	OS	42%	32%	15%	26%	35%	44%	26%	34%
	Full-System	14%	-4%	9%	0%	12%	-3%	4%	-5%

3.4 Performance Evaluation

We evaluate the benefits of integrating the above predictors with OS-aware predictions on a dynamically scheduled superscalar processor using a full-system simulator that captures OS behavior as well. Table 7 summarizes the configuration of the simulated machine architecture. We use SimOS MXS model [2], which simulates a superscalar microprocessor with multiple instruction issue, register renaming, dynamic scheduling, and speculative execution with precise exceptions. The simulated architectural model is an 8-issue superscalar processor with instruction latencies as in the MIPS R10000 [28]. By default, the branch prediction algorithm allows fetch unit to fetch through up to 4 unresolved branches. In our model, a misprediction will cause a 10-cycle penalty. BHSR is speculatively updated and later corrected after a misprediction. BHT counter update takes place in order at instruction commit time.

Figure 10 shows the IPC improvement for this scenario. Since instruction counts are the same, IPC improvement is indicative of execution cycle improvement. Results are depicted for the 13 evaluated programs. Comparison of predictors integrating OS-aware prediction techniques with Gshare, Multi-Hybrid, Agree and Bi-Mode predictors is presented. The scale of Y-axis is varied for each benchmark due to their differences in IPC. Split BHSR predictors improve IPC performance on all of the benchmarks for all of the four types of base predictors. This benefit is particularly substantial in those programs where user/OS aliasing is significant, such as *jess*, *jack*, *vortex*, and *postgres.update* (as was illustrated in

Figure 1). The same trend can be observed in programs such as *javac* and *db*. For those programs where the impact of user/OS aliasing on misprediction is less significant (for instance, *compress* and *pmake*), the integration of OS-aware techniques shows only limited improvement.

Integration of split predictor results in improvement in many cases, even though the predictor size is not much more than 50% of the baseline predictor. In most of the cases in Gshare, Multi-Hybrid and Agree predictors, despite the small size, split predictor still results in improvement. In the case of the Bi-Mode predictors, split predictor-integrated case is inferior to the baseline for 5 of 13 benchmarks. However, if one compares them to a baseline that is comparable in size (i.e., 16K BHT entries), OS-aware split predictor with 18K BHT entries (16K U-BHT + 2K K-BHT) outperforms 16K BHT entries baseline predictor in all cases, resulting up to 10% of IPC speedup [12].

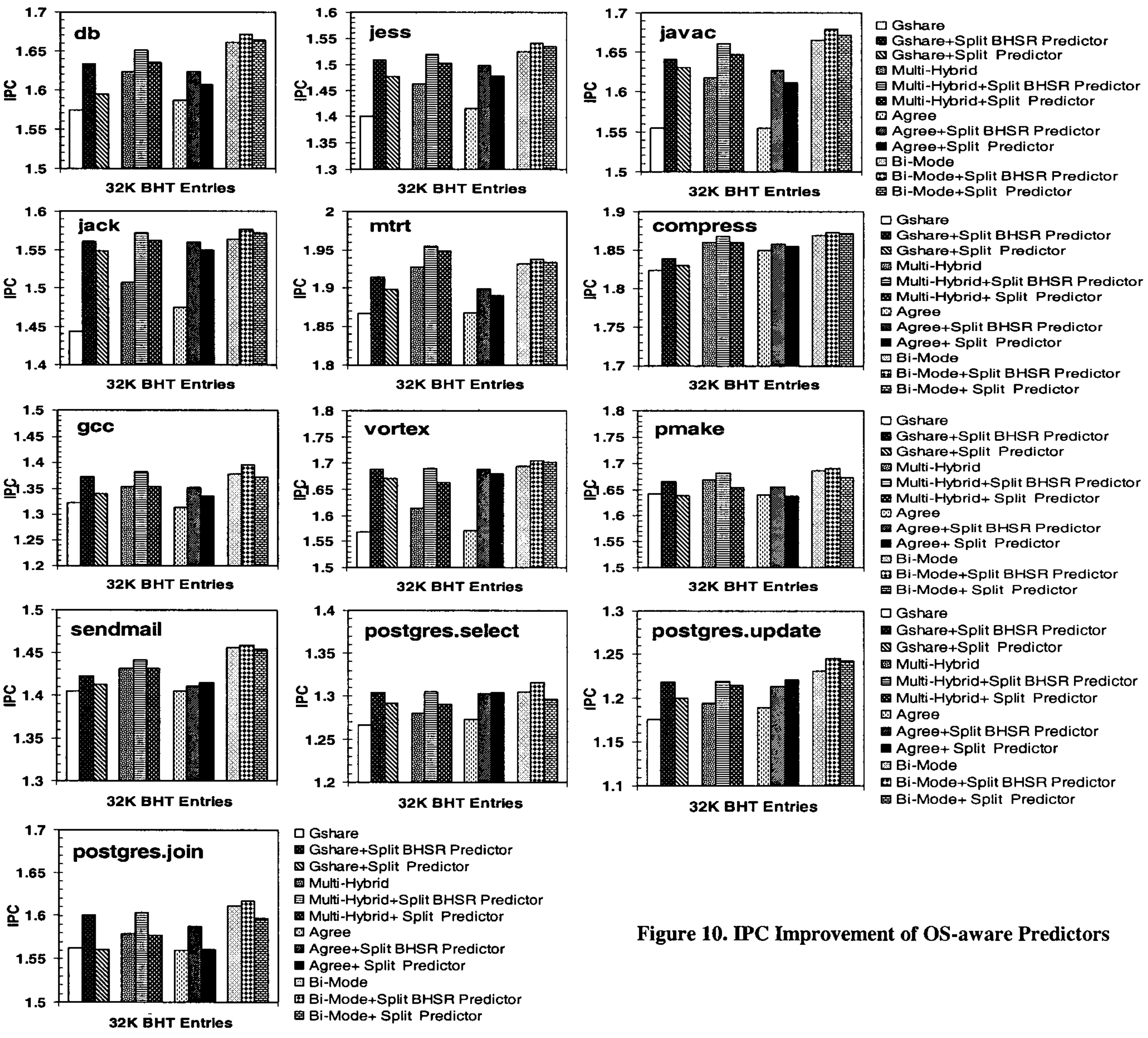


Figure 10. IPC Improvement of OS-aware Predictors

Compared with a Gshare predictor, the two proposed techniques – split BHSR predictor and split predictor yield up to 8% and 7% of IPC improvement respectively. This improvement is a result of the removal of aliasing mispredictions.

The integration of OS-aware prediction into Multi-Hybrid predictor yields up to 5% of IPC gain. As described earlier, Multi-Hybrid allocates the largest prediction resource to its Gshare component and its overall prediction accuracy is more impacted by Gshare than any other predictor. Hence, the replacement of the conventional Gshare with the proposed OS-aware Gshare predictors improves performance.

By introducing OS-aware prediction on the Agree predictor, up to 7% of IPC improvement can be achieved. The performance of Agree predictor is largely dependent on branch biases and possibility of identifying the biased behavior the first time the branch is introduced into the BTB [18]. If the branch does not show strongly biased behavior, there is still frequent aliasing between instances of a branch that do not comply with the biasing bit and instances which do comply with the biasing bit. Once we incorporate OS-aware predictions into the Agree predictor, the filtering out of the visible portion of weakly biased kernel branches leads more U-BHT entries to reach "agree" status.

Table 7. Simulated Machine Architecture

Processor Core	
Fetch/Decode/Issue/Retire Width	8
Instruction Window Size	128
Reorder Buffer Size	128
Number of Function Units	2xIssue Width
Latency of Function Units	MIPS R10000 Like
Branch Target Buffer (BTB)	2048-entry, 4-way
Return Address Stack	32-entry w/ misprediction repair
Misprediction Penalty	10 cycles
Load Store Queue Size	64
Memory Hierarchy	
MMU	Fully associative TLB, 48-entries, 4KB page size
L1 I-Cache	32KB, 2-way(LRU), 64B blocks, 4MSHRs, 2 ports, 1 cycle latency
L1 D-Cache	32KB, 2-way(LRU), 32B blocks, 4MSHRs, 2 ports, 1 cycle latency
L2 Cache	1MB, 2-way(LRU), 128B blocks, 4MSHRs, 2 ports, 10 cycle latency
Memory	256MB, 60-cycle access

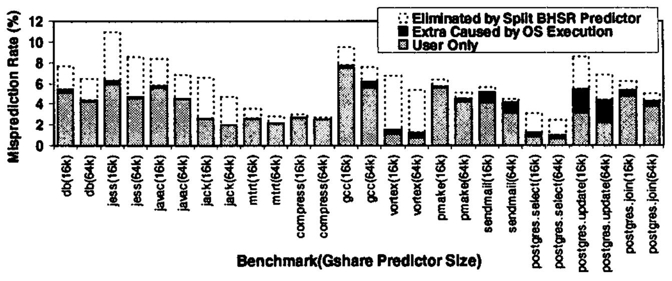
The IPC improvement of OS-aware Bi-Mode is marginal (1%), but it should be noted that the OS-aware Bi-Mode consumes only equivalent or less resource to achieve this performance enhancement. Thus, OS-aware prediction leads to the same performance with less hardware.

The results shown in Figure 10 also indicate that the combination of the OS-aware prediction and a simple predictor (for instance, Gshare) can outperform sophisticated predictors (e.g., Multi-Hybrid and Agree) with larger size configuration.

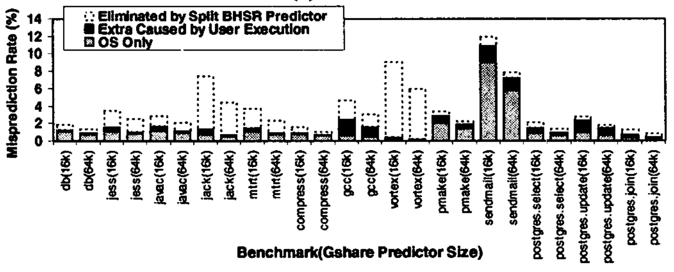
In summary, architectural support for specific OS branch behavior can enhance prediction performance without increasing predictor size or complexity. Current and next generation microprocessors are becoming increasingly sensitive to branch prediction accuracy due to the use of deeper pipelines and wider issue microarchitecture. The proposed techniques are expected to yield more ILP performance benefit on aggressive implementations with higher misprediction penalties.

3.5 Discussion

We motivated the research in this paper using Figure 1, which showed that kernel interference increases user misprediction from 1.1x to 6x (with an average of 2.1x). Similarly, we observed that user interference increases OS misprediction from 1.3x to 129x (with an average of 13x) [12]. In this subsection, we revisit this characterization in the presence of the OS-aware prediction.



(a)User



(b)OS

Figure 11. Impact of User/OS Execution on OS-aware Split BHSR Predictor

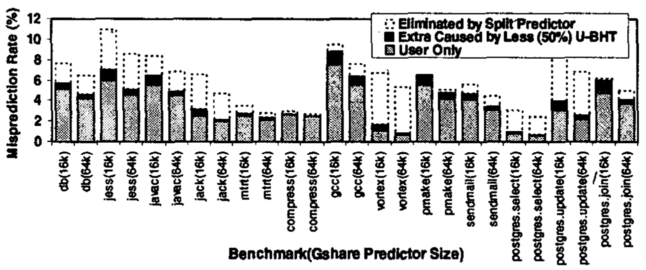
Figure 11 illustrates the impact of user/OS execution on branch prediction after OS-aware split BHSR predictor is integrated with Gshare. Compared with Figure 1, OS-aware split BHSR predictor significantly reduces the negative impact of user/OS interference on branch prediction, resulting in the drop of mispredictions from 2.1x to 1.2x and from 13x to 2x in user and OS space respectively.

As described in subsection 3.1, split BHSR predictor reduces the misprediction by providing interference free branch history for both user and kernel sides. In order to investigate the impact of the initial state of the K-BHSR when switching to the OS mode from the user mode, an experiment is performed with a variant of split BHSR predictor where the K-BHSR is cleared out upon an OS call is made. We compare the misprediction rates of a baseline split BHSR predictor with this variant of split BHSR predictor scheme. As shown in Table 8, the split BHSR predictor with K-BHSR zeroing out on an OS call provides similar performance result with that of a baseline split BHSR predictor, implying that removing the interference in the predictor state between user and kernel modes is more important than really figuring out what state to leave the predictor in when entering the kernel.

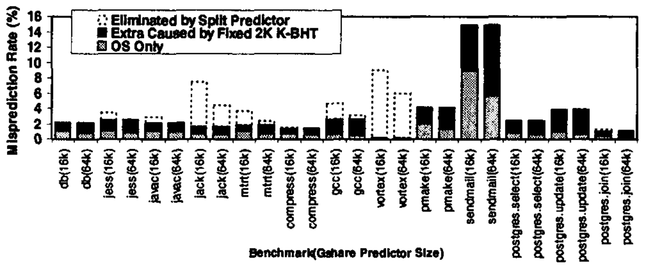
Table 8. Effect of Zeroing out K-BHSR in Split BHSR Predictor

	db	jess	javac	jack	mtrt	compress	gcc	vortex	pmake	sendmail	postgres (select)	postgres (update)	postgres (join)
Baseline	3.49	5.09	4.98	3.15	2.75	2.68	8.42	1.69	6.01	8.80	1.70	4.16	4.86
with K-BHSR Zeroing out	3.40	5.12	5.01	3.13	2.73	2.74	8.43	1.69	6.03	9.02	1.69	4.15	4.85

Similarly, Figure 12 revisits the impact of user/OS on branch misprediction after an OS-aware split predictor is integrated. Compared with Figure 1, OS-aware split predictor cost-effectively reduces the negative impact of kernel code on branch misprediction in user part. The misprediction reduction by OS interference removal outweighs the extra misprediction caused by using less (50%) BHT resource on all benchmarks except *pmake*. In the OS part, the fixed size 2K K-BHT still outperforms the performance of a unified 16K BHT on benchmarks *jess*, *javac*, *jack*, *mtrt*, *gcc* and *vortex*.



(a)User



(b)OS

Figure 12. Impact of User/OS Execution on OS-aware Split Predictor

4. SUMMARY AND CONCLUSION

Control flow prediction is one of the key issues in the design of high performance processors. It is extremely important that processor hardware, software and the operating system collaborate with each other to deliver high performance. The operating system affects control flow predictability by introducing the additional user/OS branch aliasing in predictor hardware. Compared to the branches in user code, the OS branches are usually invoked by the exception-driven and intermittently executed kernel routines and may have different biased behavior caused by performing operations not common in user mode. Thus, when interacted with user branches, the OS branches increase misprediction significantly. Current branch predictors have paid less attention to the OS requirements and therefore, do not contain mechanisms to specifically alleviate the user/OS aliasing.

This paper focuses on understanding and improving the control flow predictability in the light of the OS code. Having characterized kernel and user branch execution behavior and quantified their impact on misprediction rates of a commercial OS, we propose OS-aware branch prediction designed to reduce user/OS branch aliasing without adding extra hardware for branch de-aliasing.

The proposed OS-aware prediction is a technique that advocates orchestrating branch correlation information and/or branch history information for user and kernel branches individually. The proposed OS-aware prediction can be incorporated into many other predictors, ranging from a naïve Gshare to the more sophisticated Multi-Hybrid, Agree and Bi-Mode predictors, to further improve prediction accuracy. More precisely, on the 32K BHT entries predictors, incorporating OS-aware strategies into previously proposed Gshare, Multi-Hybrid, Agree and Bi-Mode predictors yields up to 34%, 23%, 27% and 9% prediction accuracy improvement and up to 8%, 5%, 7% and 1% execution speedup respectively. Simulation results also show that the combination of the OS-aware prediction and a simple predictor (for instance, Gshare) can outperform sophisticated predictors (e.g., Multi-Hybrid and Agree) with larger size configuration.

OS-aware techniques improve prediction performance by cost-effectively alleviating the user/OS branch aliasing. Moreover, it provides opportunities for catering user and kernel branches with differently tuned structures. For example, compared with a conventional design, the OS-aware split predictor requires access to only one of the smaller prediction tables for a given branch instruction mode (OS or user), which can result in energy savings and low-latency access. These advantages are valuable in the light of power and clock frequency constraints in emerging processor and branch predictor designs [9][17]. In our future work, we plan to model the energy consumption and the access latency of the OS-aware branch predictors.

Historically, privilege bits are used to protect system critical resources such as page tables and process control blocks for security purposes. The results of our study show that on a fine-grained resource sharing superscalar microprocessor, the protection of performance-critical microarchitecture hardware, such as branch prediction tables, also is important. It is likely that future high-end microprocessors will aggressively support more predictions (e.g. value prediction). Further research is needed to investigate the benefit of OS-aware microarchitecture on these predictors.

5. ACKNOWLEDGMENTS

We would like to thank Steven K. Reinhardt (University of Michigan), David Wood (University of Wisconsin) and all anonymous reviewers for their insightful comments and suggestions. We thank Xiaodong Zhang, Zhao Zhang and Zhichun Zhu (College of William & Mary) for their help in setting up the *postgres* database. This research is supported by the National Science Foundation under grant numbers 0113105, 9807112, 0103583, 9988164, 0073419 and CAREER 0093085, by a State of Texas Advanced Technology program grant, by a grant from GSRC, and by Tivoli, Motorola, Intel, IBM and Microsoft Corporations.

6. REFERENCES

[1] T. E. Anderson, H. M. Levy, B. N. Bershad, E. D. Lazowska, The Interaction of Architecture and Operating System Design, In Proceedings of the fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 108-120, 1991.

[2] J. Bennett and M. Flynn, Performance Factors for Superscalar Processors, Technical Report CSL-TR-95-661, Computer Systems Laboratory, Stanford University, Feb. 1995.

[3] P. Chang and U. Banerjee, Profile-guided Multi-heuristic Branch Prediction, In Proceedings of the International Conference on Parallel Processing, 1995.

[4] P. Y. Chang, M. Evers, and Y. Patt, Improving Branch Prediction Accuracy by Reducing Pattern History Table Interference, In Proceedings of International Conference on Parallel Architectures and Compilation Techniques, pages 48-57, 1996.

[5] K. Diefendorff, HP, Intel Complete IA-64 Rollout, Microprocessor Report, pages 1-9, Apr. 2000.

[6] A. N. Eden and T. Mudge, The YAGS Branch Prediction Scheme, In Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture, pages 69-77, 1998.

[7] 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.

[8] 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.

[9] D. A. Jiménez, S. W. Keckler, and C. Lin, The Impact of Delay on the Design of Branch Predictors, In Proceedings of the 33rd Annual International Symposium on Microarchitecture, 2000.

[10] 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.

[11] 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.

[12] T. Li, L. K. John, A. Sivasubramaniam, N.Vijaykrishnan and J. Rubio, Understanding and Improving Operating System Effects in Control Flow Prediction, Technical Report, Department of Electrical and Computer Engineering, University of Texas at Austin, June 2002. http://www.ece.utexas.edu/projects/ece/lca/ps/tao-TR-june-2002.pdf.

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

[14] S. McFarling, Combining Branch Predictors, WRL Technical Note TN-36, Digital Equipment Corporation, June 1993.

[15] P. Michaud, A. Seznec and R. Uhlig, Trading Conflict and Capacity Aliasing in Conditional Branch Predictors, In Proceedings of the 24th International Symposium on Computer Architecture, pages 292-303, 1997.

[16] J. Ousterhout, Why aren't Operating Systems Getting Faster as Fast as Hardware?, In Proceedings of the Summer 1990 USENIX Conference, pages 247-256, 1990.

[17] Parikh, K. Skadron, Y. Zhang, M. Barcella, and M. Stan, Power Issues Related to Branch Prediction, In Proceedings of 8th International Symposium on High Performance Computer Architecture, 2002.

[18] C. Perleberg and A. Smith, Branch Target Buffer Design and Optimization, IEEE Transactions on Computers, 42(4): pages 396-412, 1993.

[19] "PostgreSQL", http://www.us.postgresql.org/.

[20] 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.

[21] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, A. Gupta, The Impact of Architectural Trends on Operating System Performance, In Proceedings of the 15th ACM Symposium on Operating System Principles, pages 285-298, 1995.

[22] 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.

[23] 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.

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

[25] 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.

[26] M. Stonebraker, L. A. Rowe and M. Hirohama, The Implementation of Postgres, IEEE Transactions on Knowledge and Data Engineering, 2(1), March 1990.

[27] Transaction Processing Council, The TPC-C Benchmark, http://www.tpc.org/tpcc/.

[28] K. C. Yeager, MIPS R10000, IEEE Micro, vol.16, no.1, pages 28-40, Apr. 1996.

[29] T. Y. Yeh and Y. N. Patt, Two-Level Adaptive Branch Prediction, In Proceeding of 24th International Symposium on Microarchitecture, pages 51-61, 1991.

[30] 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.

[31] 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.

[32] Y. Zhang, J. Zhang, A. Sivasubramaniam, C. Liu and H. Franke, Characterizing TPC-H on a Clustered Database Engine from the OS Perspective, In Proceedings of the Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW-02), 2002.