Published in the Proc. of the 23rd Annual International Symposium on Computer Architecture, May 1996.

An Analysis of Dynamic Branch Prediction Schemes on System Workloads

Nicolas Gloy, Cliff Young, J. Bradley Chen, and Michael D. Smith
Division of Applied Sciences, Harvard University

{ng, cyoung, bchen, smith}@eecs.harvard.edu

Abstract

Recent studies of dynamic branch prediction schemes rely almost exclusively on user-only simulations to evaluate performance. We find that an evaluation of these schemes with user and kernel references often leads to different conclusions. By analyzing our own Atom-generated system traces and the system traces from the Instruction Benchmark Suite, we quantify the effects of kernel and user interactions on branch prediction accuracy. We find that user-only traces yield accurate prediction results only when the kernel accounts for less than 5% of the total executed instructions. Schemes that appear to predict well under user-only traces are not always the most effective on full-system traces: the recently-proposed two-level adaptive schemes can suffer from higher aliasing than the original per-branch 2-bit counter scheme. We also find that flushing the branch history state at fixed intervals does not accurately model the true effects of user/kernel interaction.

Keywords: branch prediction, correlation, 2-level adaptive prediction, system traces

1 Introduction

With the explosion of new superscalar microarchitectures, there has been a mounting pressure on microprocessor architects to improve the predictability of the conditional branches in the program flow. With the trend toward larger branch misprediction penalties due to the use of deeper pipelines, breaks in the program flow can quickly throttle the performance of these wide-issue microprocessors. Several recent studies [11, 14, 20] have proposed new hardware branch prediction schemes to address this problem. To date, the evaluation of these new techniques has been done almost exclusively with user-level traces of applications such as those found in the SPEC92 benchmark suite. This study goes beyond that work to use full-system traces (i.e. traces with user and kernel references) to evaluate the effectiveness of several two-level adaptive branch prediction schemes. This study also analyzes the performance of these dynamic branch prediction schemes on kernel-intensive applications such as an HTTP server and those found in the IBS benchmark suite [18].

All dynamic branch prediction schemes in this study are similar in that they use a table of two-bit, up-down, saturating counters. A 2-bit counter summarizes the past outcomes of a branch stream, using this information to predict the next branch outcome [10, 17]. The method of selection of a 2-bit counter in this table defines the type of dynamic branch prediction implemented. We evaluate four dynamic schemes that have been shown to be particularly successful at predicting user-level branches: simple per-branch dynamic [17], GAs [14, 21], gshare [11], and PAs [21]. The last three schemes are two-level adaptive schemes which exploit patterns in the recent local or global branch history to improve prediction accuracy.

While recent studies have demonstrated the benefit of two-level adaptive schemes on benchmarks such as SPEC92, Young et al. [23] point out some potential problems with these approaches as the number of static branches to predict increases. Since a large number of programs in the SPEC92 benchmark suite contain very few static branch sites, these benchmarks do not stress the size of the hardware branch prediction tables in most studies. We evaluate two-level adaptive schemes on larger applications, such as those found in the Instruction Benchmark Suite (IBS) [18]. Since these benchmarks do not cover the entire spectrum of applications, we also evaluate the two-level adaptive schemes using our own system traces. We gathered these traces with the Atom tool-building system [5]. Overall, our Atom traces include a selection of the SPEC92 benchmarks and several large, system-intensive applications like an HTTP server. Unlike the SPEC92 benchmarks, the HTTP server spends a significant amount of its execution time in kernel routines. In summary, through the use of the IBS traces and our own system traces, we are able to analyze the performance of two-level adaptive branch prediction schemes under three operating systems and on a wide spectrum of applications.

Different workloads spend different amounts of time in user and kernel code. We find that user-level traces of applications that spend the vast majority of their time in user code (for example the SPEC92 benchmarks) give good approximations of overall prediction accuracy. However, the prediction accuracy on benchmarks with an even relatively small amount of kernel activity (just 10% of instructions) is not modeled well by user-only traces. Schemes that appear the best in user-only traces (e.g. gshare with a large branch history depth) do not always perform best on full-system traces. Our results show that including kernel branches in the branch trace can greatly increase the number of static branches predicted and thus worsen the effects of aliasing. The negative effect of aliasing on prediction accuracy is more pronounced in the two-level schemes with large history depths than in locally-oriented schemes that rely on small history depths [16]. We also find that flushing the branch history state [13, 15] at fixed intervals does not accurately model the true effects of user/kernel interactions: some schemes are more sensitive than others to periodic flushing.

Section 2 summarizes the recent advances in branch prediction, and it describes the major reasons for poor prediction accuracy in a dynamic branch prediction scheme. Section 3 presents our simulation methodology and our benchmark applications. Section 4 discusses our experimental results. Section 5 presents the conclusions of this work.

2 Understanding Branch Prediction Schemes

In the last five years, researchers have made steady improvements in the accuracy of static and dynamic branch prediction schemes by exploiting the relationships between program branches and the patterns of behavior of individual branches. To understand the operation and to compare the performance of these schemes, Young et al. [23] introduced an analytical framework for today's branch prediction schemes. Figure 1 summarizes the main components of that framework. As illustrated by this figure, the recently proposed branch prediction schemes predict the future outcome of a branch by accessing a *predictor* which summarizes some portion of the past outcome of this branch. For example, most dynamic branch prediction schemes summarize the past history of a branch through the use of a simple finite-state machine implemented as a 2-bit, up/down, saturating counter. The *divider* in Figure 1 selects the predictor, e.g. a 2-bit counter, used for each prediction. Before 1991, the divider in the best branch prediction schemes chose a predictor based on the address of the branch to predict [10, 12, 17]. The dynamic versions of these schemes maintained a table of 2-bit counters, referred to as a branch history table (BHT), indexed by the branch address. Figure 2a illustrates the hardware for this approach, which we refer to as *2bc*.

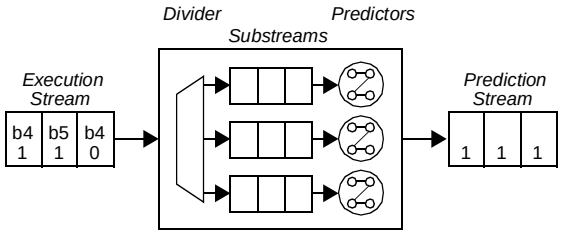

Figure 1. Framework for describing a branch prediction scheme [23]. The divider mechanism splits the program execution stream into substreams, each of which is predicted by a single predictor.

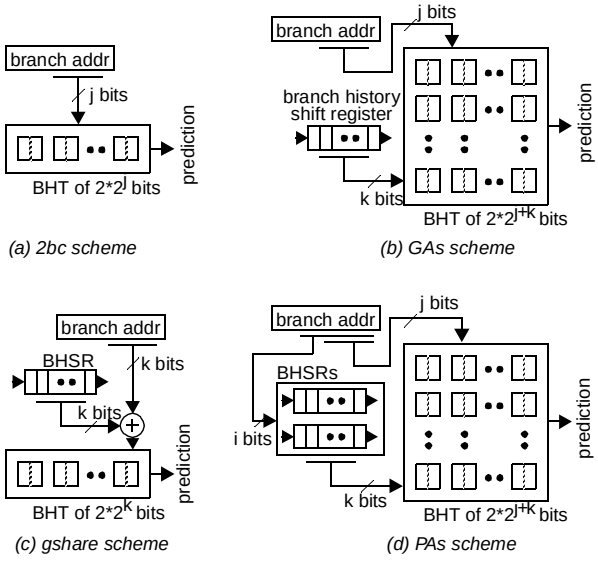

Figure 2. Block diagrams for the four dynamic branch prediction schemes evaluated in this study. The branch history table (BHT) comprises an array of 2-bit, up/down, saturating counters used as predictors.

Recently, several researchers have empirically shown that we can improve branch prediction accuracy by building more elaborate divider mechanisms [11, 14, 20]. By appropriately dividing a program's dynamic branch stream into many substreams, we can produce substreams that are more predictable. For dynamic schemes, Yeh and Patt [20] introduced the concept of "two-level adaptive" branch prediction schemes whose dividers include branch history shift registers (BHSRs) which record the recent directions of program branches. Their divider mechanisms use the contents of these shift registers in addition to branch address information to create highly predictable substreams. Krall [9] and Young and Smith [22] describe code transformations that yield similar results for static branch prediction approaches.

In this paper, we focus on three two-level adaptive branch prediction schemes that have been shown to be effective on user-level code [11, 14, 21]. Figure 2 depicts each of these schemes. The first is called *GAs*, and it uses a single, global BHSR to record the outcome of the past *k* branches. As discussed by Pan, So, and Rahmeh [14], GAs exploits the *correlation* between branch executions in a program; correlation occurs when the outcome of one or more branch executions helps to determine the outcome of a future branch. GAs chooses a 2-bit counter from the BHT by concatenating the contents of the global BHSR with the current branch address. McFarling [11] proposes a modification to this scheme where the BHSR contents are exclusive-or-ed with the branch address. McFarling refers to this new scheme as *gshare*. The exclusive-or function permits the use of longer history and more address bits for a fixed size BHT than GAs. Ideally, this extra information results in more substreams that are more predictable. The final two-level adaptive branch prediction scheme that we consider is called *PAs* [21]. The PAs scheme maps each program branch into a table of BHSRs; the contents of the selected BHSR are concatenated to a portion of the branch address to select a 2-bit counter from the BHT. This scheme exploits repeating patterns in the execution of a single program branch (e.g. loop branches that iterate a constant number of times), but not correlation between distinct static branches. GAs and gshare may be able to capture some of the same looping patterns as PAs on short loop branches, but their use of global history prevents them from exploiting patterns in longer loops.

The analysis performed by Young et al. [23] suggests that the prediction accuracies generated by the current implementation of dynamic prediction schemes like those in Figure 2 are negatively affected by problems of *aliasing* and *training overhead*. Aliasing occurs when the hardware divider assigns streams from different branches to the same 2-bit counter. Though the intermingling of the individual branch streams can *constructively*, *destructively*, or *neutrally* impact the prediction accuracy of the individual branches, Young et al. showed that destructive aliasing occurs more frequently and with larger magnitude than constructive aliasing, especially if the working set of the application is large or the BHT is small in size. Training overhead refers to the fact that a 2-bit counter needs to be "primed" for a particular conditional branch by observing a few executions of that branch. Young et al. did not discuss the effects of training overhead in detail, but this effect is observable in some of their shorter benchmark runs. For these runs, the schemes with finer dividers did not always achieve better prediction accuracies than simpler schemes because the training overhead of many substreams became a noticeable percentage of the total number of predictions. In a simple scheme with a small number of substreams, the few predictions done during 2-bit counter training amounts to a negligible number of mispredictions. As we will see in Section 4, the problem of aliasing can become even more pronounced for traces of system activity.

Though there have been several studies exploring the effects of system references on instruction cache performance, the vast majority of the work in branch prediction has focused on user-only traces. Nair [13] and Perleberg and Smith [15] attempt to model the effects of context switches on the user-level component of branch misprediction by regularly flushing the BHT during a user-only trace. We are familiar with only one study that has performed branch prediction simulations with system-level traces. The study by Lee and Smith [10] contains three traces of the MVS operating system executing a commercial workload. Since this work occurred before the invention of two-level adaptive branch prediction schemes, Lee and Smith report only the performance of these traces on a 2bc scheme (in addition to other 2bc-like schemes).

3 Methodology

We use trace-driven simulation of user and kernel activity to evaluate prediction accuracy on a range of branch prediction techniques. We use traces that were collected by two different measurement systems, one hardware-based and the other software-based, on three different operating systems. By using traces from two independent sources, we can benefit from the complementary advantages of hardware and software systems and achieve a higher overall degree of confidence in the quality of our simulations.

For our first set of traces, we used the IBS traces from the University of Michigan [18]. These traces were generated on a DECstation 3100 with a MIPS R2000 processor. The traces are designed to provide a realistic instruction reference stream, overcoming limitations of benchmark suites such as SPEC92 which fit in most on-chip instruction caches and do not induce significant operating system activity. IBS contains traces for two different operating systems, ULTRIX from Digital Equipment Corporation [19] and Mach 3.0 from Carnegie Mellon University [1, 3].

For our simulations, we used the following items from the IBS trace record: the memory address referenced; the flag that indicates whether the reference was to instruction or data space; the flag that indicates user or kernel mode; and the opcode fetched by an instruction reference. With this information, we generated a branch stream (as illustrated in Figure 1) that we used as input for our branch prediction scheme simulator. The top of Table 1 gives a description of the IBS benchmarks. Table 2 presents some general statistics for each IBS trace.

We collected additional traces using Atom [5] on an Digital AlphaStation 400/233 running Digital Unix (formally OSF-1), release 3.2. With the Atom tool-building system, it is possible to instrument both user programs and the Digital Unix kernel, thereby collecting complete data for the simulation with no special-purpose hardware and no source-code modifications to the operating system.

When software-based measurements of system activity are used for architectural simulation, care must be taken to avoid excessive distortion in measured behavior due to the overhead of the measurement system. Two kinds of distortion occur: space dilation and time dilation [2]. To remove space dilation effects, we ran our experiments on a machine with enough physical memory so additional system activity due to virtual memory effects did not occur. To compensate for time dilation, we scaled clock interrupts via a software technique, warmed the file system buffer cache before any measured runs, avoided disk-bound workloads (where branch misprediction penalties do not matter anyway), and used a dedicated network. A more detailed discussion of time dilation in Atom tools and techniques for avoiding their ill effects can be found in a separate technical report [4].

Benchmark			Abbr.	Description
IBS	Mach 3.0 and Ultrix	groff	[mu].groff	GNU C++ implementation of ‘nroff’; version 1.09
		gs	[mu].gs	Ghostscript version 2.4.1, single page of text and graphics
		jpeg_play	[mu].jpeg	‘xloadimage’ version 3.0; displays two JPEG images
		mpeg_play	[mu].mpeg	‘mpeg_play’ version 2.0, 85 frames from compressed file
		nroff	[mu].nroff	Ultrix 3.1 version of nroff
		real_gcc	[mu].gcc	GNU C compiler, version 2.6
		sdet	[mu].sdet	Multiprocess benchmark from the SPEC SDM suite
		verilog	[mu].vlog	Verilog-XL version 1.6b; simulating a microprocessor
		video_play	[mu].video	modified ‘mpeg_play’; 610 frames from uncompressed file
SPEC92	Digital Unix	026.compress	o.co	‘ref.in’ input (1M file size)
		008.espresso	o.es	‘bca.in’ input
		gcc	o.gc	compiles ‘tree.i’ from its own source, version 2.6.3
		022.li	o.li	shortened version of ‘ref.lsp’ input
		072.sc	o.sc	‘load1’ input
		089.su2cor	o.su	‘short.in’ input
Other		Andrew bench	o.ab	Andrew without the compilation part
		ghostscript	o.gs	version 2.6.1; displays a 10-page conference paper; includes X server
		HTTP daemon	o.ht	NCSA httpd version 1.4; 8 server processes; 1000 requests
		mpeg_play	o.mp	Version 2.2; does not use shared memory; includes X server

Table 1: Description of our benchmark programs. We replaced the SPECint92 version of gcc with version 2.6.3 because we had trouble compiling it on our Alpha machines. The descriptions of the IBS benchmarks are based on those provided in [18].

The bottom of Table 1 includes a brief description of the benchmarks traced under Atom. Table 3 presents the statistics for these benchmarks. They represent a range of applications with differing degrees of kernel and user activity. Most of the recent previous work in branch prediction has focused on the SPEC92 benchmark suite. We chose a sample of these benchmarks, and as shown in Table 3, they spend very little of their total instruction count in the kernel. In addition to these benchmarks, we evaluated four benchmarks chosen for their high level of kernel activity.

From the Atom trace, we construct a branch stream that is identical in format to the stream produced from the IBS benchmarks. We then feed this stream to our simulator code. Though Section 4 presents the results from only a single simulation run of each benchmark, we ran each benchmark in Table 3 three times to verify the stability of our results. We found that the maximum difference in the prediction accuracy between two runs with the same branch prediction scheme was always less than 0.3% (typically less than 0.1%). For a particular benchmark, the prediction accuracy difference between schemes and sizes was always much greater than the difference between runs.

4 Experimental Results

Our experiments concentrated on two basic questions: are the simulation results of user-level traces representative of the prediction accuracy of a dynamic branch prediction scheme on a full-system trace, and does periodic flushing of the BHT during a user-level

Benchmark			Dynamic instruction count		Dynamic branch count		Protection boundary crossings
			user	kernel	user	kernel	
IBS	Mach 3.0	m.groff	105.3M (91%)	10.6M (09%)	11.8M (94%)	0.8M (06%)	25990
		m.gs	102.9M (86%)	17.1M (14%)	14M (91%)	1.4M (09%)	86778
		m.jpeg	69.8M (77%)	21.2M (23%)	11.8M (87%)	1.7M (13%)	177212
		m.mpeg	68.7M (62%)	42.1M (38%)	5.2M (55%)	4.3M (45%)	232144
		m.nroff	124.2M (92%)	11M (08%)	21.8M (95%)	1.1M (05%)	58018
		m.gcc	113.6M (90%)	12.5M (10%)	15.3M (93%)	1.2M (07%)	20008
		m.sdet	12.2M (28%)	30.8M (72%)	1.6M (38%)	2.6M (62%)	108562
		m.vlog	41.9M (81%)	9.6M (19%)	5.6M (88%)	0.8M (13%)	24760
		m.video	25.2M (45%)	30.2M (55%)	2.7M (48%)	2.9M (52%)	190402
	Ultrix	u.groff	90.7M (86%)	14.2M (14%)	10.5M (88%)	1.4M (12%)	7054
		u.gs	85.7M (86%)	14.1M (14%)	12.3M (88%)	1.6M (12%)	8836
		u.jpeg	138.9M (92%)	12.2M (08%)	20.6M (93%)	1.6M (07%)	12170
		u.mpeg	77.8M (78%)	21.6M (22%)	6.8M (72%)	2.7M (28%)	14804
		u.nroff	119.4M (92%)	10.9M (08%)	21.1M (93%)	1.5M (07%)	7034
		u.gcc	93.7M (87%)	13.7M (13%)	12.9M (90%)	1.4M (10%)	7858
		u.sdet	0.6M (01%)	41.5M (99%)	0.1M (02%)	5.4M (98%)	5744
		u.vlog	36.4M (77%)	10.6M (23%)	5.4M (87%)	0.8M (13%)	7470
		u.video	15.8M (30%)	36.7M (70%)	1.8M (32%)	3.9M (68%)	14470

Table 2: Basic statistics for the IBS benchmark programs. “Protection boundary crossings” counts the number of times that the trace switches from a user process to kernel plus the switches from the kernel to a user process. Please note that the UNIX server is a user process under Mach and its activity is counted in the user categories.

trace accurately reflect the effect of kernel branches on the user-level component of prediction accuracy? Sections 4.1 and 4.2, respectively, discuss our findings for these two questions.

Throughout this section, we report the mispredict rates for prediction schemes with hardware state of 4K bits, 16K bits, and 64K bits.[1] We refer to a particular scheme with the identifier “name.size”, where “size” is the number of bits in that scheme’s hardware state. For example, the identifier “2bc.4K” indicates that the simulator used the hardware “2bc” branch prediction scheme with a BHT size of 2K 2-bit counters. The smallest hardware sizes correspond roughly to the amount of branch prediction hardware found in today’s microprocessors [6, 7]. We chose the largest scheme size because it has the same number of storage bits as an 8-kilobyte cache.

1. For 2bc, GAs, and gshare, these sizes correspond to BHTs with 2K, 8K, and 32K 2-bit counters. We omit the relatively small hardware costs of the BHSRs on the GAs and gshare schemes. We assumed that two BHSR bits have the same hardware cost as one 2-bit counter for the PAs schemes.

Benchmark			Dynamic instruction count		Dynamic branch count		Protection boundary crossings
			user	kernel	user	kernel	
Digital Unix	SPEC92	o.co	70.9M (95%)	3.5M (05%)	12.1M (95%)	0.6M (05%)	6462
		o.es	332.3M (98%)	6.2M (02%)	60.0M (98%)	1.4M (02%)	34388
		o.gc	145.7M (97%)	4.3M (03%)	30.6M (98%)	0.8M (02%)	18890
		o.li	121.4M (96%)	4.6M (04%)	32.5M (97%)	0.9M (03%)	32072
		o.sc	81.8M (91%)	7.8M (09%)	20.4M (93%)	1.6M (07%)	26360
		o.su	385.2M (99%)	2.2M (01%)	10.7M (96%)	0.4M (04%)	5842
	Other	o.ab	18.0M (38%)	29.2M (62%)	4.2M (44%)	5.3M (56%)	11870
		o.gs	183.0M (88%)	24.0M (12%)	29.6M (85%)	5.1M (15%)	28112
		o.ht	53.2M (23%)	180.3M (77%)	10.6M (21%)	39.0M (79%)	170138
		o.mp	92.7M (87%)	13.6M (13%)	6.6M (71%)	2.7M (29%)	10636

Table 3: Basic statistics for our Digital Unix benchmark programs. “Protection boundary crossings” counts the number of times that the trace switches from a user process to kernel plus the switches from the kernel to a user process.

Scheme	Branch address bits used for		BHSR bits used for BHT index (k)	Size of scheme (Kbits)
	BHSR selection (i)	BHT index (j)		
2bc.4K	0	11	0	4
2bc.16K	0	13	0	16
2bc.64K	0	15	0	64
pas.4K	10	6	3	4
pas.16K	12	8	3	16
pas.64K	14	10	3	64
gas.4K	0	4	7	4
gas.16K	0	5	8	16
gas.64K	0	6	9	64
gsh.4K	0	11	11	4
gsh.16K	0	13	13	16
gsh.64K	0	15	15	64

Table 4: Specific parameters used for each dynamic branch prediction scheme evaluated in this study. The address bits used are the lower i/j bits of the branch’s word address. The BHT size is $2^{\max(j,\,k)}$ bits for gshare schemes and 2^{j+k} bits for other schemes.

Table 4 lists the specific parameters of each hardware scheme. The gas.4K entry matches the size and organization of the branch prediction hardware in the NexGen Nx586 [7]. The larger GAs schemes were chosen by scaling up the NexGen parameters.[2] For PAs, we initially experimented with an organization that corresponded to the reported parameters used by the branch prediction unit in the Pentium Pro processor (i = 9, j = 9, k = 4) [6]. However, this organization did not achieve misprediction rates as low as the PAs configurations in Table 4. Similarly, PAs implementations

2. One might be tempted to run a set of GAs simulations to determine the best trade-off between j and k parameters for our benchmarks. However, as will be seen later in this section, the best “GAs” scheme for some of our benchmarks has a k value of zero (i.e. a 2bc scheme). Hence, we use the NexGen design parameters as a reasonable starting point for experiments.

Benchmark			Count of static branches in percentile								Count of static branches touched		
			50%		90%		95%		99%				
			user	full	user	full	user	full	user	full	user	kernel	full
IBS	Mach 3.0	m.groff	106	115	383	497	558	813	1219	2069	6678	2590	9268
		m.gs	84	100	864	1095	1430	1874	3181	4177	12488	2379	14867
		m.jpeg	3	4	44	157	159	405	647	1373	1812	1711	3523
		m.mpeg	70	78	625	1006	1037	1576	2128	2908	7140	2473	9613
		m.nroff	25	27	171	237	297	457	732	1254	5654	2305	7959
		m.gcc	435	444	3307	3589	4892	5381	8383	9403	17038	2316	19354
		m.sdet	63	115	711	1221	1198	1990	2471	3578	4825	2703	7528
		m.vlog	77	99	612	941	912	1434	1825	2918	5673	2495	8168
		m.video	12	31	649	999	1077	1575	1958	2765	4816	2309	7125
	Ultrix	u.groff	108	122	356	454	485	690	889	1658	3344	2989	6333
		u.gs	137	156	911	1113	1227	1633	2476	3772	9521	3144	12665
		u.jpeg	3	3	27	57	52	179	153	898	5558	2328	7886
		u.mpeg	36	64	140	490	195	785	666	1899	2589	3009	5598
		u.nroff	27	30	183	227	273	374	516	966	2487	2762	5249
		u.gcc	316	322	2786	3057	4134	4685	7416	8698	14516	2845	17361
		u.sdet	27	8	418	467	598	817	1231	1920	1928	3382	5310
		u.vlog	74	92	551	802	842	1281	1491	2387	2720	1916	4636
		u.video	27	75	146	710	252	1037	619	1736	1724	2882	4606
SPEC92	Digital Unix	o.co	4	5	13	16	16	19	18	249	375	4505	4880
		o.es	16	17	145	178	251	318	523	652	3221	4354	7575
		o.gc	349	372	3072	3261	4590	4930	8531	9587	21397	5446	26843
		o.li	23	24	71	81	95	122	147	266	1108	4254	5362
		o.sc	56	66	279	469	418	748	734	1219	2923	5081	8004
		o.su	5	6	26	32	36	63	79	445	1937	4210	6147
Other		o.ab	6	17	19	783	28	1318	100	2983	1124	5564	6688
		o.gs	108	145	509	844	854	1411	1769	3172	8235	6549	14784
		o.ht	13	82	242	988	371	1413	648	2373	2094	7679	9773
		o.mp	16	25	74	352	111	684	282	2103	2129	6076	8205

Table 5: Counts of static branches touched in our benchmarks during the trace along with counts of the minimum number of static branches accounting for 50%, 90%, 95%, and 99% of the dynamic branch executions.

with the same bit cost and a longer history depth (k = 6) also performed worse than the selected PAs schemes at the 4K and 16K sizes.

The SPEC92 benchmarks have been criticized because a significant portion of their dynamic branch execution count is due to a very small numbers of static branches. The data in Table 5 supports this claim. Only o.gc, o.sc, and possibly o.es exercise a significant number of user static branches in the 95% column. The same statement can be made of the distribution of branch executions in the *full* (kernel and user) trace. Table 5 also provides proof that our non-SPEC benchmarks are a challenging workload for the sizes of our branch prediction schemes. Except for a few benchmarks (e.g. u.jpeg and u.nroff), it takes well over 400 static branches to account for 95% of the dynamic branch executions in the full-system trace. Often, the user branches alone are a large portion of this total.

4.1 Predicting user and kernel branches

Researchers have evaluated two-level adaptive branch prediction schemes using single-process traces of user-only activity. Their studies repeatedly concluded that the addition of extra hardware to exploit more specific patterns in the branch stream, as found in gshare or PAs, achieved better branch prediction accuracies than the simpler hardware found in the GAs or 2bc schemes. These results have encouraged researchers to develop even more elaborate hardware and convinced microprocessor vendors to implement these two-level adaptive schemes in new superscalar processors [6, 7, 8]. The results in this section show that the mispredict rate obtained from a trace of user-level branches is often a poor indicator of a scheme's mispredict rate on user and kernel branches. We also find that 2bc can provide the lowest mispredict rate in some cases.

Figure 3 summarizes the results of our simulations, counting how many times each scheme showed the best mispredict rate at a particular scheme size, for user-only and full-system traces. Even from this summary, we can make a number of interesting observations. First, the best dynamic prediction scheme for a trace of user-only branches is not always the same as the best scheme for a full-system trace of user and kernel branches. Second, for both user-only and full-system traces, a scheme like gshare that uses a long branch history predicts better as the scheme size increases. The PAs scheme is the best prediction scheme at the small (4K) scheme size; PAs and GAs do best at the middle (16K) scheme size, while GAs and gshare are best at the large (64K) scheme size. To make the same point a different way, the use of long branch histories appears to penalize schemes at small scheme sizes. In addition, the inclusion of kernel branches appears to have a similar effect to that of decreasing the size: schemes with shorter histories do better. As we

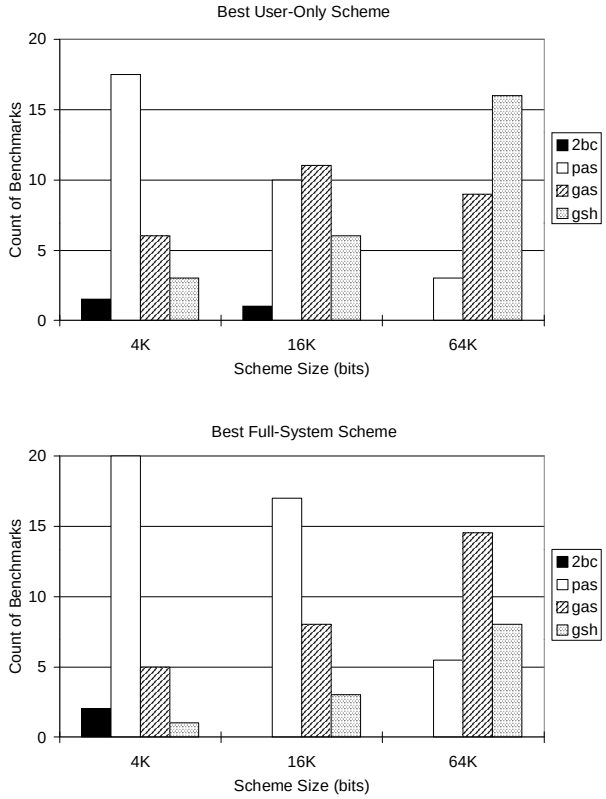

Figure 3. Distribution of benchmark simulations grouped by the scheme that yields the lowest mispredict rate. The top and bottom plots show results from user-only and full-system traces respectively. This figure summarizes data in Tables 8, 9, and 10.

will show later in this section, these observed trends are due to the effects of aliasing [23].

Tables 8, 9, and 10 (at the end of the paper) present the user-only and full-system mispredict rates for each benchmark under our range of 2bc, PAs, GAs, and gshare schemes. The data in these tables demonstrates that a mispredict rate as measured in a simulation of user-only activity is not necessarily a reliable indicator of the true mispredict rate on the full-system trace. For example, u.video at gas.64K achieves a user-only mispredict rate of 1.17% while the full-system trace mispredict rate is 3.71%; more than a factor of three worse. Fortunately, the user-only mispredict rates for the SPEC92 benchmarks under Digital Unix match fairly well with the mispredict rates achieved under full-system tracing, providing some credibility to the results of previous studies. As illustrated by the results for o.sc, the match becomes worse as the scheme size decreases or as the history depth of the prediction scheme increases. Furthermore, as the percentage of total instructions executed in user mode decreases, the user-only mispredict rate for user-only experiments quickly deviates from that achieved under full-system tracing. This observation makes sense intuitively, provided that the kernel-only and user-only mispredict rates differ.

The scatter plots in Figures 4 and 5 (on the next two pages) depict the distortions introduced in the mispredict rate by eliminating kernel references from our branch prediction simulations. For each chart, the horizontal axis shows the user-only mispredict rate, while the vertical axis shows the full-system mispredict rate. If user-only traces accurately represent the full-system mispredict rate, then we would expect all points to appear on the diagonal. As expected, this is true for the SPEC benchmarks (the solid circles in Figures 4 and 5). The IBS and the Other Digital Unix benchmarks show some significant deviations from the diagonal, although these deviations decrease with larger scheme sizes.

Benchmark group	Scheme	Scheme Size (bits)		
		4K	16K	64K
IBS	2bc	0.05	0.03	0.04
	pas	0.10	0.08	0.06
	gas	0.12	0.11	0.09
	gsh	0.12	0.12	0.10
SPEC92	2bc	0.01	0.01	0.01
	pas	0.03	0.02	0.02
	gas	0.02	0.01	0.01
	gsh	0.03	0.02	0.01
Other Digital Unix	2bc	0.07	0.09	0.10
	pas	0.20	0.16	0.10
	gas	0.10	0.03	0.07
	gsh	0.17	0.12	0.06

Table 6: Arithmetic mean of distortion for each benchmark group. Distortion is calculated using the formula $(|u-f|) / (u+f)$, where u is the user-only mispredict rate and f is the full-system mispredict rate. A value of 0 means no distortion while a value of 1 means that one rate dwarfs the other.

Focusing on Figure 4, the 2bc graphs appear similar across all scheme sizes. This suggests that the 2bc scheme approaches the point of diminishing returns at BHT sizes of 4K bits; enlarging the table does not significantly reduce the mispredict rate since little aliasing is occurring. This is intuitively borne out by the static branch percentiles in Table 5; few of our benchmarks use more than 4K static branches at the 95th percentile of static branches. However, the IBS benchmarks, which tend to have smaller overall mispredict rates and thus are less visually striking in the scatter plots, still show reasonable improvements from larger scheme sizes.

In Figures 4 and 5, the two-level adaptive schemes do not appear to reach the point of diminishing returns for the scheme sizes that we examined. With increasing scheme size, each graph looks like a scaled-down version of its predecessor, corresponding to better overall mispredict rates from the larger sizes. It also appears that the deviations from the diagonal decrease with larger sizes. Both of these trends make intuitive sense, since the larger sizes should reduce aliasing within the combined set of user and kernel branches.

To quantify our intuitions of reduced distortions at larger scheme sizes, we examined the normalized distortion, the difference between user-only and full-system mispredict rates divided by their sum. This metric ranges from 0 to 1, where 0 indicates no distortion, and 1 indicates that one of the mispredict rates is a tiny fraction of the other (high distortion). Table 6 summarizes this metric for each grouping of benchmark, scheme, and size. The distortion for the SPEC benchmarks is always below 0.03. The distortion for the IBS benchmarks hovers around 0.1, while the Other Digital Unix benchmarks have a slightly higher distortion ranging from 0.1 to 0.2. As we suspected from visual examination of Figures 4 and 5, the distortion generally decreases with larger scheme sizes. It sometimes increases due to the fact that we measure the distortion between the user-only mispredict rate and the full-system mispredict rate, while several of the IBS and the Other Digital Unix benchmarks are dominated by kernel and not user branches.

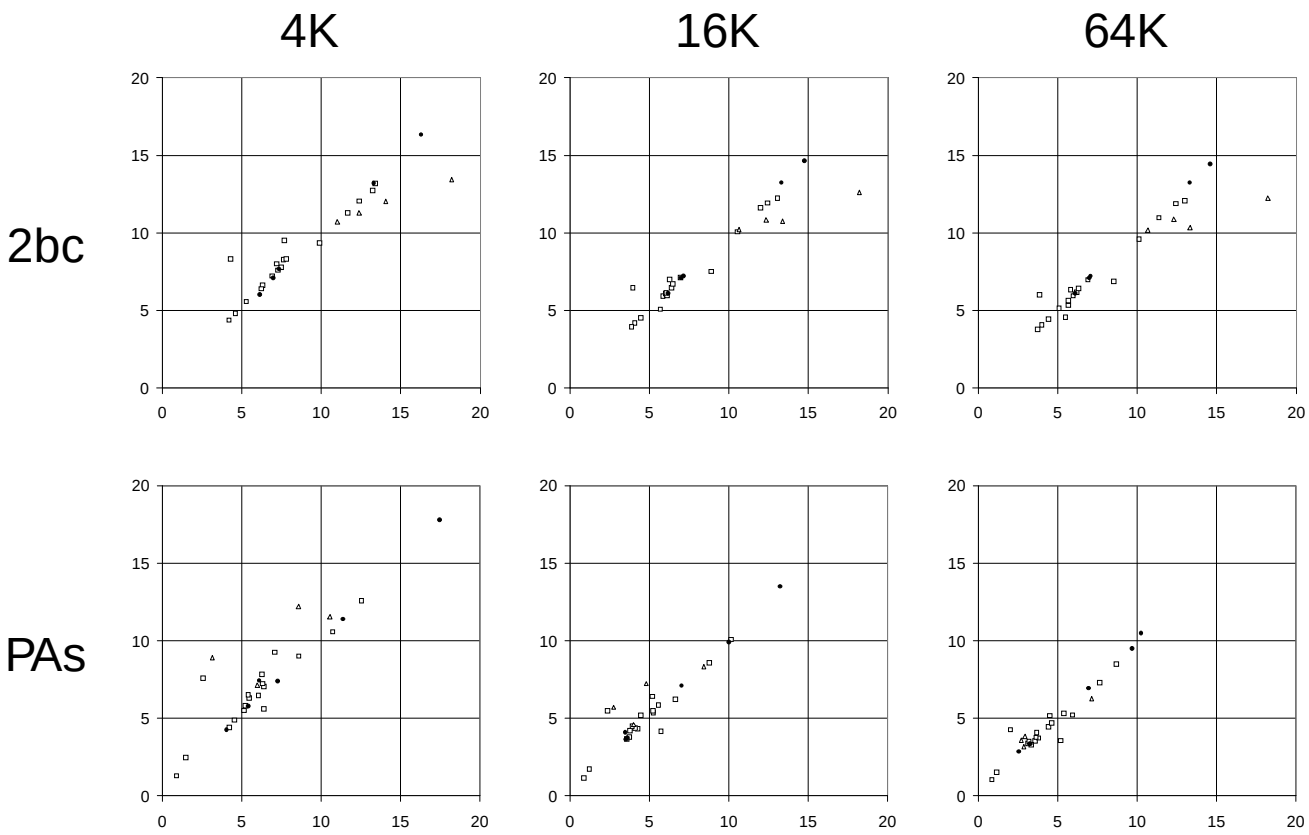

Figure 4. Scatter plots comparing user-only mispredict rates (horizontal axis) with full-system traces (vertical axis) for 2bc schemes (top) and PAs schemes (bottom). Hollow squares show IBS benchmark results; solid circles show SPEC92 results, and hollow triangles show the Other Digital Unix results. The 2bc graphs omit results for o.co, which are off the scale.

The addition of kernel branches to the simulation has increased aliasing (contention) in the prediction scheme hardware. Table 11 presents the full-system mispredict rates where all aliasing (both BHSR and BHT address aliasing) has been removed. Unsurprisingly, unaliased mispredict rates are always better than the mispredict rate of the corresponding scheme in our study. As we observed earlier, for the same scheme, smaller sizes suffer more aliasing than larger sizes. This is borne out by the larger differences in mispredict rates of unaliased and practical implementation at smaller scheme sizes. For example, under gas.4K, o.ht shows a mispredict rate of 10.27%, while the equivalent $k = 7$ unaliased GAs scheme achieves 2.65%. Aliasing adds almost 300% more mispredictions. The gas.64K scheme shows a mispredict rate of 3.78%; the unaliased $k = 9$ GAs scheme achieves a rate of 2.36%. Aliasing adds just 60% more mispredictions. Similarly, schemes with deeper branch histories suffer more aliasing than schemes with shallow branch histories. For example, m.video under gas.16K shows a mispredict rate of 7.26%, while the equivalent $k = 8$ unaliased GAs scheme achieves 2.79%. Aliasing adds almost 160% more mispredictions. The gsh.16K scheme shows a mispredict rate of 7.51%; the corresponding unaliased $k = 13$ GAs scheme[3] achieves a rate of 2.03%. Aliasing adds 270% more mispredictions.

From our data, it appears that a user-only mispredict rate accurately reflects the full-system mispredict rate if the percentage of the total instruction count spent in user mode is greater than 95%. If the percentage is less than 90%, the results of a user-only trace cannot be trusted. The 90–95% range is a grey area. The reverse of this observation is demonstrated by u.sdet, which spends less than 2% of the total instruction count in user mode. In this case, we found that the mispredict rate of the kernel-only trace is a good indicator of the full-system mispredict rate.

4.2 Simulating the effect of kernel branches

To date, very few branch prediction studies have considered the effects of user/kernel interaction on prediction accuracy. Nair [13] and Perleberg and Smith [15] each attempt to model the effects of context switching on the user-only branch mispredict rates by flushing the BHT at a fixed interval of instructions. This method is inexact, because interactions with the kernel or other processes do not necessarily flush the branch history state. A short switch may have little effect on the state, and a large table may suffer lower contention and thus suffer less ill effect. Nair, Perleberg, and Smith use traces that omit system activity, and hence they are not able to verify the true effects of kernel branches on the user-only component of the mispredict rate. To evaluate the validity of this approach, we modified our simulator to flush the BHTs and BHSRs at fixed intervals of instructions during a user-only trace. We then compared the resulting mispredict rates to the user component of the full-system trace simulation. Some flush interval will produce the same mispredict rate as the user component of the full-system trace; we call this number the *effective flush interval* (EFI). If flushing at fixed intervals is an accurate methodology, then we would expect the EFI to remain constant across different prediction scheme organizations and sizes.

3. Unaliased gshare schemes are GAs schemes of the same history depth.

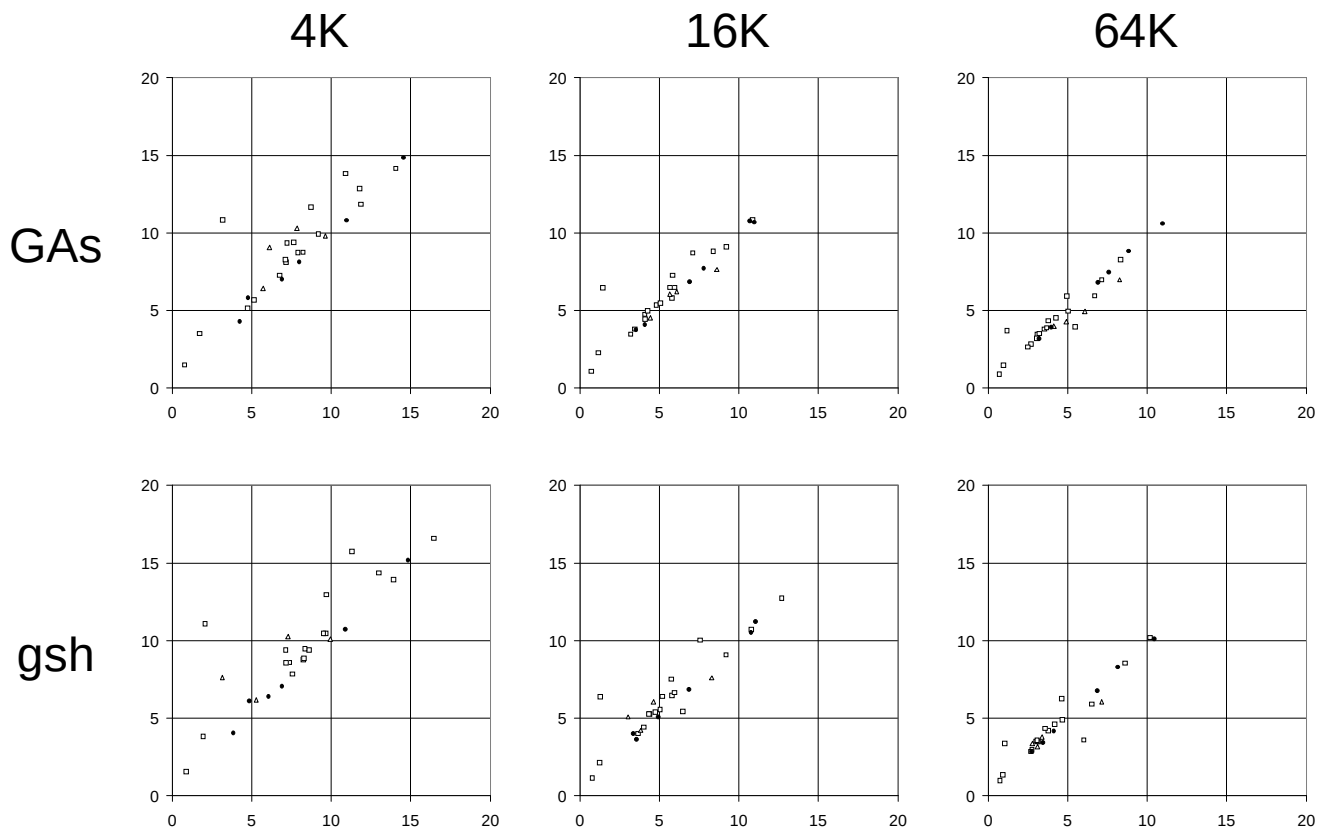

Figure 5. Scatter plots comparing user-only mispredict rates (horizontal axis) with full-system traces (vertical axis) for GAs schemes (top) and gshare schemes (bottom). Hollow squares show IBS benchmark results; solid circles show SPEC92 results, and hollow triangles show the Other Digital Unix results.

Our full-system traces include both user and kernel references. So, we can evaluate the accuracy of periodic flushing as a methodology for estimating the effect of user and kernel contention on the branch prediction hardware. It suffices to show one example where periodic flushing produces inaccurate and misleading results. Our u.video benchmark is such an example. This benchmark spends more than two-thirds of its time in the kernel, so one might think that the "pollution" of the branch prediction scheme caused by kernel branches would resemble flushing.

Table 7 compares the full-system mispredict rate and the mispredict rates generated by periodic flushing for u.video. We can see two crucial problems. First, under each scheme, the EFI increases with increasing scheme size. This means that periodic flushing cannot be used to compare different scheme sizes for the same scheme, because it can overly penalize the larger sizes. The 2bc entries at a flush interval of 10,000 instructions give a concrete example of this: the periodic flushing results imply that larger 2bc tables result in only small improvements in prediction accuracy. But the user component of the full-system trace shows significant improvements with increasing scheme size. Larger scheme sizes remove aliasing between user and kernel branches in this case. Since periodic flushing provides no model for the *other* branches contending for the table, it cannot model the benefits from reduced aliasing.

The second important problem is that the EFI for u.video varies between prediction schemes even at a constant scheme size. This makes periodic flushing a useless methodology for comparing different branch prediction schemes. The scheme with the larger EFI will be unfairly penalized by the effects of periodic flushes. For example, using a flush interval of 10,000 instructions to compare 64K-bit implementations would lead one to believe that PAs gives the best mispredict rate, followed by GAs, 2bc, and gshare. The order ordering from the full-system trace simulation is different: gshare and GAs achieve the best mispredict rates, followed by PAs and 2bc.

The previous discussion proves that we cannot trust the numerical values produced by periodic flushing. Table 7 demonstrates that we cannot even trust the overall trends implied by periodic flushing results. Using a flush interval of 10,000 instructions, periodic flushing reports that gshare predicts *worse* with increasing scheme size—exactly the opposite of the truth.

5 Conclusions

Using full-system (i.e. combined user/kernel) traces gives realistic results that lead to different conclusions about the effectiveness of existing dynamic branch prediction schemes than do the results from user-only traces. We find that including kernel references often increases aliasing, and this effect may cause schemes with short branch histories to achieve better prediction accuracies than those with deep branch histories. While SPEC92 is user-dominated (so prior work in branch prediction retains value), system designers and customers probably want to match their test workloads to a wider range of user/kernel mixes. Simulations that ignore kernel activity risk dangerous inaccuracy: elaborate two-level schemes that appear good under user-only traces may turn out to be less attractive when the whole system is considered. These problems appear even

Size and Scheme		EFI	Mispredict rate (%)			
			Full-system trace	Periodic flush interval		
				10K	100K	1M
4K	2bc	90-100K	5.19	6.69	5.13	4.36
	PAs	50-60K	5.36	6.68	4.80	3.29
	GAs	20-30K	5.39	6.62	4.51	3.42
	gshare	10-20K	5.05	6.21	3.67	2.39
16K	2bc	300-400K	4.26	6.58	4.93	4.05
	PAs	200-300K	3.30	6.03	4.27	2.69
	GAs	100-200K	2.87	5.63	3.17	1.79
	gshare	100-200K	3.00	6.41	3.43	1.72
64K	2bc	1.1-1.2M	3.92	6.55	4.85	3.94
	PAs	1.7-1.8M	2.22	5.79	3.88	2.37
	GAs	600-700K	1.71	5.81	3.14	1.58
	gshare	1.4-1.5M	1.45	6.57	3.42	1.58

Table 7: Comparison of the user component of the mispredict rate from a full-system trace with the mispredict rates derived from periodic flushing intervals. The benchmark in these simulations is u.video. The effective flush interval is the periodic flush interval that achieves the same mispredict rate as the full-system trace simulation.

worse for small scheme sizes. As a rule of thumb, if both the kernel and the user account for more than 5% of the instruction mix, then combined system and kernel traces should be used.

Flushing at fixed intervals poorly models the effect of kernel branches on dynamic branch prediction schemes. It is misleading to use periodic flushing to compare different schemes with the same amount of hardware or to compare the same scheme with varying amounts of hardware. More specifically, periodic flushing fails to capture differences in the organization and size of schemes. It assumes the same amount of contention exists in a 4K-bit scheme as in a 64K-bit scheme. And, it assumes the same amount of contention in a 2bc scheme as in a gshare scheme of the same hardware size. These underlying fallacies in the periodic flushing model will persist and yield inaccurate results when periodic flushing is used to model multitasking workloads.

6 Acknowledgments

This research was sponsored in part by grants from Digital Equipment, Hewlett-Packard, and Intel. Cliff Young is funded by a Graduate Fellowship from the Office of Naval Research and an IBM Cooperative Fellowship. Brad Chen is supported by an NSF Career Award, grant number CCR-9501365. Michael D. Smith is supported by a National Science Foundation Young Investigator award, grant number CCR-9457779.

7 References

[1] M. Accetta, et al. “Mach: A New Kernel Foundation for Unix Development,” *Proc. Summer 1986 USENIX Conf.*, Jul. 1986.

[2] J. Chen. “Software Methods for System Address Tracing,” *Proc. Fourth Workshop on Workstation Operating Systems*, Oct. 1993.

[3] J. Chen and B. Bershad. “The Impact of Operating System Structure on Memory System Performance,” *Proc. 14th ACM Symp. on Operating System Principles*, Dec. 1993.

[4] J. Chen and A. Eustace. “Kernel Instrumentation Tools and Techniques,” Technical Report 26-95, Center for Research in Computing Technology, Harvard University, Cambridge, MA, Nov. 1995.

[5] A. Eustace and A. Srivastava. “ATOM: A Flexible Interface for Building High Performance Program Analysis Tools”. *Proc. Winter 1995 USENIX Technical Conf. on UNIX and Advanced Computing Systems*, Jan. 1995

[6] L. Gwennap. “New Algorithm Improves Branch Prediction,” *Microprocessor Report*, 9(4):17–21, Mar. 27, 1995.

[7] L. Gwennap. “Pentium Competitors Go Head to Head,” *Microprocessor Report*, 9(8):16, Jun. 19, 1995.

[8] L. Gwennap. “Nx868 Goes Toe-to-Toe with Pentium Pro,” *Microprocessor Report*, 9(14):8, Oct. 23, 1995.

[9] A. Krall. “Improving Semi-static Branch Prediction by Code Replication,” *Proc. ACM SIGPLAN ’94 Conf. on Prog. Lang. Design and Implementation*, Jun. 1994.

[10] J. Lee and A. Smith. “Branch Prediction Strategies and Branch Target Buffer Design,” *Computer*, 17(1), Jan. 1984.

[11] S. McFarling. “Combining Branch Predictors,” *WRL Technical Note TN-36*, June 1993.

[12] S. McFarling and J. Hennessy. “Reducing the Cost of Branches,” *Proc. of 13th Annual Intl. Symp. on Computer Architecture*, Jun. 1986.

[13] R. Nair. “Dynamic Path-Based Branch Correlation,” *Proc. 28th Annual Intl. Symp. on Microarchitecture*, Nov. 1995.

[14] S. Pan, K. So, and J. Rahmeh. “Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation,” *Proc. 5th Annual Intl. Conf. on Architectural Support for Prog. Lang. and Operating Systems*, Oct. 1992.

[15] C. Perleberg and A. Smith. “Branch Target Buffer Design and Optimization,” *IEEE Transactions on Computers*, 42(4):396-412, Apr. 1993.

[16] S. Sechrest, C. Lee, and T. Mudge, “The Role of Adaptivity in Two-Level Adaptive Branch Prediction,” *Proc. 28th Annual Intl. Symp. on Microarchitecture*, Nov. 1995.

[17] J. Smith. “A Study of Branch Prediction Strategies,” *Proc. 8th Annual Intl. Symp. on Computer Architecture*, Jun. 1981.

[18] R. Uhlig, D. Nagle, T. Mudge, S. Sechrest, and J. Emer. “Instruction Fetching: Coping with Code Bloat,” *Proc. 22nd Annual Intl. Symp. on Computer Architecture*, Jun. 1995.

[19] ULTRIX Documentation Group. *ULTRIX Documentation Overview for RISC Processors*, Digital Equipment Corporation, 1989.

[20] T. Yeh and Y. Patt. “Two-Level Adaptive Branch Prediction,” *Proc. 24th Annual ACM/IEEE Intl. Symp. and Workshop on Microarchitecture*, Nov. 1991.

[21] T. Yeh and Y. Patt. “A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History,” *Proc. 20th Annual Intl. Symp. on Computer Architecture*, May 1993.

[22] C. Young and M. Smith. “Improving the Accuracy of Static Branch Prediction Using Branch Correlation,” *Proc. 6th Annual Intl. Conf. on Architectural Support for Prog. Lang. and Operating Systems*, Oct. 1994.

[23] C. Young, N. Gloy, and M. Smith. “A Comparative Analysis of Schemes for Correlated Branch Prediction,” *Proc. 22nd Annual Intl. Symp. on Computer Architecture*, Jun. 1995.

Benchmark	Scheme	Scheme Size (bits) 4K user	4K full	16K user	16K full	64K user	64K full
m.groff	2bc	6.31	6.63	5.85	5.93	5.09	5.15
	PAs	6.05	6.48	5.24	5.38	4.42	4.47
	GAs	7.90	8.74	4.80	5.35	3.54	3.81
	gsh	9.65	10.48	5.79	6.47	3.78	4.21
m.gs	2bc	7.27	7.59	6.38	6.46	6.17	6.16
	PAs	6.38	7.04	5.21	5.48	4.62	4.71
	GAs	9.18	9.93	5.94	6.48	4.25	4.52
	gsh	9.51	10.48	5.95	6.64	4.18	4.61
m.jpeg	2bc	13.23	12.72	13.05	12.23	12.98	12.07
	PAs	1.45	2.48	1.20	1.74	1.13	1.52
	GAs	1.71	3.52	1.16	2.28	0.96	1.48
	gsh	1.94	3.83	1.24	2.12	0.92	1.34
m.mpeg	2bc	9.90	9.34	8.89	7.51	8.53	6.88
	PAs	8.58	9.02	6.61	6.22	5.93	5.22
	GAs	11.80	12.86	8.38	8.82	6.69	5.95
	gsh	12.99	14.36	9.19	9.09	6.51	5.92
m.nroff	2bc	4.60	4.83	4.08	4.19	4.00	4.06
	PAs	4.52	4.89	3.72	3.81	3.55	3.56
	GAs	5.14	5.69	3.45	3.80	2.67	2.84
	gsh	8.24	8.77	4.01	4.42	2.74	2.97
m.gcc	2bc	13.38	13.19	11.98	11.64	11.37	10.99
	PAs	12.52	12.58	10.13	10.09	8.68	8.51
	GAs	14.05	14.15	10.86	10.84	8.33	8.29
	gsh	16.45	16.59	12.68	12.73	10.18	10.20
m.sdet	2bc	7.67	9.52	6.28	6.99	5.81	6.34
	PAs	7.06	9.26	5.18	6.41	4.49	5.18
	GAs	10.91	13.83	7.09	8.71	4.94	5.94
	gsh	11.31	15.75	7.56	10.04	4.61	6.24
m.vlog	2bc	7.46	7.79	6.02	6.09	5.67	5.65
	PAs	5.47	6.31	3.76	4.22	3.12	3.39
	GAs	7.14	8.11	4.08	4.75	3.08	3.44
	gsh	7.37	8.59	4.39	5.27	2.99	3.52
m.video	2bc	7.61	8.28	6.11	5.97	5.67	5.33
	PAs	6.25	7.83	4.45	5.20	3.67	4.09
	GAs	8.71	11.67	5.83	7.26	3.78	4.34
	gsh	9.68	12.95	5.76	7.51	3.57	4.34

Table 8: User-only and full-system mispredict rates for the Mach 3.0 IBS benchmarks. For each benchmark, we highlight the lowest overall mispredict rate for each set of scheme simulations.

Benchmark	Scheme	Scheme Size (bits) 4K user	4K full	16K user	16K full	64K user	64K full
u.groff	2bc	5.29	5.58	4.46	4.53	4.41	4.45
	PAs	5.15	5.52	4.25	4.33	3.78	3.74
	GAs	6.74	7.27	4.10	4.45	3.05	3.20
	gsh	8.28	8.85	5.04	5.56	3.13	3.47
u.gs	2bc	6.91	7.22	6.05	6.14	5.97	5.96
	PAs	5.22	5.81	4.08	4.38	3.62	3.78
	GAs	8.22	8.75	5.07	5.49	3.67	3.89
	gsh	8.61	9.38	4.75	5.40	3.07	3.52
u.jpeg	2bc	12.4	12.06	12.42	11.93	12.41	11.90
	PAs	0.87	1.30	0.85	1.15	0.84	1.05
	GAs	0.77	1.49	0.70	1.07	0.68	0.88
	gsh	0.86	1.56	0.76	1.14	0.72	0.96
u.mpeg	2bc	7.19	8.01	6.94	7.13	6.88	6.97
	PAs	6.32	7.24	5.56	5.87	5.39	5.32
	GAs	7.64	9.39	5.68	6.48	5.03	4.96
	gsh	7.13	9.40	5.19	6.40	4.65	4.90
u.nroff	2bc	4.20	4.38	3.88	3.95	3.72	3.77
	PAs	4.20	4.43	3.57	3.66	3.31	3.30
	GAs	4.73	5.16	3.19	3.48	2.49	2.65
	gsh	7.54	7.84	3.65	4.01	2.69	2.85
u.gcc	2bc	11.68	11.29	10.53	10.07	10.11	9.61
	PAs	10.72	10.59	8.77	8.57	7.63	7.30
	GAs	11.88	11.85	9.21	9.10	7.13	6.98
	gsh	13.92	13.92	10.77	10.73	8.60	8.55
u.sdet	2bc	6.23	6.39	5.69	5.08	5.49	4.56
	PAs	6.37	5.62	5.73	4.16	5.19	3.57
	GAs	7.21	9.34	5.79	5.82	5.48	3.95
	gsh	8.34	9.47	6.46	5.45	6.01	3.60
u.vlog	2bc	7.79	8.33	6.45	6.71	6.32	6.43
	PAs	5.40	6.50	3.91	4.49	3.18	3.52
	GAs	7.12	8.27	4.25	4.98	3.20	3.52
	gsh	7.15	8.56	4.33	5.27	3.06	3.58
u.video	2bc	4.29	8.32	3.95	6.45	3.84	6.02
	PAs	2.55	7.59	2.34	5.48	2.01	4.27
	GAs	3.16	10.83	1.42	6.47	1.17	3.71
	gsh	2.06	11.08	1.27	6.38	1.04	3.38

Table 9: User-only and full-system mispredict rates for the Ultrix IBS benchmarks. For each benchmark, we highlight the lowest overall mispredict rate for each set of scheme simulations.

Benchmark	Scheme	Scheme Size (bits) 4K user	4K full	16K user	16K full	64K user	64K full
o.co	2bc	25.74	24.82	25.74	24.82	25.74	24.84
	PAs	10.98	10.78	10.97	10.65	10.97	10.57
	GAs	10.89	10.73	10.80	10.51	10.46	10.10
	gsh	11.38	11.37	10.02	9.91	9.70	9.48
o.es	2bc	6.14	6.00	6.17	6.08	6.11	6.08
	PAs	4.28	4.29	4.12	4.07	3.97	3.90
	GAs	3.88	4.03	3.58	3.62	3.44	3.42
	gsh	4.07	4.24	3.52	3.61	3.28	3.32
o.gc	2bc	16.32	16.31	14.75	14.63	14.59	14.42
	PAs	14.57	14.82	10.69	10.76	8.85	8.79
	GAs	14.86	15.16	11.06	11.21	8.16	8.26
	gsh	17.48	17.80	13.24	13.47	10.28	10.48
o.li	2bc	13.33	13.19	13.33	13.20	13.33	13.21
	PAs	8.01	8.12	7.82	7.70	7.60	7.44
	GAs	6.09	6.38	4.94	5.06	4.16	4.16
	gsh	5.41	5.77	3.59	3.71	3.24	3.33
o.sc	2bc	7.36	7.64	7.15	7.20	7.09	7.18
	PAs	4.80	5.82	3.55	3.74	3.22	3.16
	GAs	4.87	6.08	3.38	3.98	2.75	2.83
	gsh	6.13	7.40	3.50	4.09	2.55	2.82
o.su	2bc	7.00	7.08	7.00	7.06	7.00	7.06
	PAs	6.92	6.99	6.91	6.82	6.90	6.78
	GAs	6.92	7.04	6.87	6.84	6.86	6.76
	gsh	7.26	7.39	7.05	7.07	6.95	6.91
o.ab	2bc	18.21	13.43	18.21	12.60	18.22	12.26
	PAs	6.12	9.06	6.09	6.21	6.08	4.95
	GAs	3.15	7.61	3.03	5.08	2.77	3.37
	gsh	3.14	8.91	2.73	5.72	2.69	3.58
o.gs	2bc	11.01	10.73	10.64	10.21	10.66	10.17
	PAs	5.71	6.42	4.43	4.52	4.13	3.98
	GAs	5.28	6.16	3.82	4.22	3.09	3.15
	gsh	5.98	7.12	4.00	4.60	2.85	3.17
o.ht	2bc	14.05	12.04	13.37	10.77	13.32	10.34
	PAs	7.84	10.29	5.66	6.05	4.91	4.27
	GAs	7.28	10.27	4.63	6.04	3.38	3.78
	gsh	8.56	12.22	4.80	7.23	2.92	3.84
o.mp	2bc	12.37	11.30	12.33	10.84	12.31	10.89
	PAs	9.63	9.81	8.61	7.63	8.26	6.98
	GAs	9.94	10.11	8.29	7.60	7.13	6.05
	gsh	10.54	11.57	8.42	8.33	7.12	6.27

Table 10: User-only and full-system mispredict rates for the Digital Unix benchmarks. For each benchmark, we highlight the lowest overall mispredict rate for each set of scheme simulations

Benchmark	Unaliased GAs of history depth (k) 0	7	8	9	11	13	15	Unaliased PAs $k = 3$
m.groff	5.06	3.19	3.00	2.82	2.57	2.40	2.38	4.09
m.gs	6.02	3.61	3.42	3.21	2.83	2.70	2.64	4.23
m.jpeg	12.03	1.21	1.14	1.04	0.95	0.89	0.91	1.34
m.mpeg	6.73	4.57	4.24	4.00	3.47	3.30	3.30	4.55
m.nroff	4.02	2.72	2.47	2.36	2.22	2.17	2.15	3.43
m.gcc	10.76	5.83	5.49	5.16	4.82	4.62	4.45	7.63
m.sdet	6.11	3.84	3.63	3.49	3.28	3.06	3.06	4.34
m.vlog	5.50	3.08	2.81	2.74	2.54	2.39	2.35	2.96
m.video	5.21	2.95	2.79	2.46	2.15	2.03	2.02	3.39
u.groff	4.34	2.80	2.63	2.45	2.24	2.12	2.09	3.43
u.gs	5.86	3.67	3.52	3.11	2.62	2.21	2.05	3.25
u.jpeg	11.89	0.82	0.80	0.80	0.78	0.77	0.78	0.99
u.mpeg	6.92	5.24	4.47	4.29	3.83	3.73	3.68	5.03
u.nroff	3.76	2.58	2.34	2.21	2.08	2.04	2.01	3.21
u.gcc	9.40	5.09	4.80	4.50	4.24	4.12	4.00	6.58
u.sdet	4.50	3.23	2.78	2.68	2.53	2.37	2.37	3.08
u.vlog	6.34	3.13	2.87	2.83	2.56	2.46	2.37	3.12
u.video	5.92	3.32	2.56	2.45	2.18	2.01	1.97	3.71
o.co	11.94	10.05	9.98	9.69	9.67	9.41	9.26	10.51
o.es	4.68	3.34	3.31	3.27	3.10	3.04	3.03	3.85
o.gc	10.02	5.46	5.10	4.72	4.29	4.00	3.85	7.00
o.li	10.61	3.79	3.60	3.31	2.99	2.74	2.79	6.48
o.sc	3.95	2.55	2.45	2.36	2.26	2.16	2.12	2.74
o.su	6.87	6.70	6.70	6.70	6.71	6.72	6.72	6.73
o.ab	4.66	2.78	2.69	2.51	2.41	2.37	2.33	3.63
o.gs	6.19	2.97	2.77	2.65	2.37	2.22	2.15	3.49
o.ht	5.36	2.65	2.44	2.36	2.16	1.99	1.91	3.18
o.mp	8.11	6.38	6.19	5.57	5.31	5.23	5.13	6.29

Table 11: Full-system mispredict rates where all aliasing (both BHSR and BHT address aliasing) has been removed.