Performance issues in correlated branch prediction schemes

This paper is available from the Center for Research in Computing Technology, Division of Applied Sciences, Harvard University as technical report TR-23-95.

Performance Issues in Correlated Branch Prediction Schemes

Nicolas Gloy, Michael D. Smith, Cliff Young
Division of Applied Sciences
Harvard University, Cambridge, MA 02138
{ng, smith, cyoung}@das.harvard.edu

Abstract

Accurate static branch prediction is the key to many techniques for exposing, enhancing, and exploiting Instruction Level Parallelism (ILP). The initial work on static correlated branch prediction (SCBP) demonstrated improvements in branch prediction accuracy, but did not address overall performance. In particular, SCBP expands the size of executable programs, which negatively affects the performance of the instruction memory hierarchy. Using the profile information available under SCBP, we can minimize these negative performance effects through the application of code layout and branch alignment techniques. We evaluate the performance effect of SCBP and these profile-driven optimizations on instruction cache misses, branch mispredictions, and branch misfetches for a number of recent processor implementations. We find that SCBP improves performance over (traditional) per-branch static profile prediction. We also find that SCBP improves the performance benefits gained from branch alignment. As expected, SCBP gives larger benefits on machine organizations with high mispredict/misfetch penalties and low cache miss penalties. Finally, we find that the application of profile-driven code layout and branch alignment techniques (without SCBP) can improve the performance of the dynamic correlated branch prediction techniques.

1 Introduction

Recent work in branch prediction [3, 18, 25, 30, 31, 32, 33] has led to the development of both hardware and software schemes that achieve high prediction accuracy by exploiting branch correlation. The motivation for this work stems from the fact that the performance of superscalar and deeply pipelined processors can benefit significantly from a small improvement (a couple of percent points) in prediction accuracy. As with any technique however, there is a point of diminishing returns where the incremental costs of the technique begin to outweigh the further improvements. In static correlated branch prediction (SCBP) techniques [33], the cost of better prediction accuracy is code expansion, and thus, the point of diminishing returns is defined primarily by the relationship between the changes in the average access time of the instruction memory subsystem and the cycle count changes enabled by the improvements in prediction accuracy. In this paper, we explore this relationship to determine how parameters such as the pipeline structure and the cache organization of a processor affect the viability of SCBP techniques. Using this framework, we also examine the performance benefits of two profile-driven optimizations on gshare [18], a dynamic branch prediction scheme that efficiently exploits branch correlation.

SCBP improves prediction accuracy for a particular branch by creating multiple copies of that branch, effectively encoding information on the outcome of previous branches in the program counter. It is clear that this will affect the instruction cache behavior of the resulting program. First, by enlarging the cache footprint of the program, SCBP increases the number of compulsory and capacity misses. Second, by increasing the code size of individual procedures and causing these procedures to shift relative to each other in memory, it can significantly change the number of conflict misses.

Though Young and Smith [33] present code expansion numbers in their initial paper on SCBP, they do not quantify the magnitude of the performance effects of this code expansion on the memory subsystem, and hence it is difficult to say when further improvements in prediction accuracy are outweighed by the memory system penalties due to greater code expansion. In this paper, we consider the first-order effects on program performance: total cycles due to branch mispredictions, branch misfetches, primary cache misses, and additional dynamic instructions due to code layout considerations. As defined by Calder and Grunwald [2], a misfetch penalty refers to any penalty associated with a correctly predicted branch. Since SCBP is a software scheme, the cycle time of the processor is not

affected and hence that component of the performance equation is unchanged. The true performance benefit of SCBP is dependent upon the effectiveness of compile-time optimizations, such as global instruction scheduling [15, 17] and code optimization [4], since these optimizations attempt to reduce the total number of cycles required to execute a program. Since the design and evaluation of sophisticated compile-time optimizations is an endless task, we do not attempt to give a definitive answer to the question of how much SCBP can ultimately improve application performance.

Even in this limited study, comparing the cache behavior of different versions of the same program can be problematic unless measures are taken to minimize unnecessary conflict misses. We have observed that small changes in the relative locations of different parts of a program can cause very significant changes in the number of cache misses, and these fluctuations can completely obscure the relation between code expansion and changes in the miss rate. Fortunately, the profiling information that is necessary to implement SCBP is sufficient for the implementation of code layout algorithms that keep instruction fetch penalties to a minimum. In particular, we have implemented three previously published techniques [24] that use profiling information to optimize the instruction cache behavior and minimize the instruction misfetch penalties of a program. We apply these techniques together with the code transformations that implement SCBP. For the vast majority of our benchmarks on all of our machine microarchitectures, SCBP gives better performance than per-branch profiled static branch prediction, and a large component of the overall benefits comes from the code layout optimizations. In fact, we find that, in addition to our SCBP scheme, the profile-driven code layout optimizations also help to improve the performance of a dynamic branch prediction scheme.

Section 2 reviews the previous work in code expanding optimizations, and it relates this work to the domain of branch prediction. Section 3 introduces our experimental methodology, and it briefly describes the code layout optimizations used in this study. Section 4 presents the results of our simulations. We conclude with a summary of our findings in Section 5.

2 Previous work

In the last five years, interest in branch prediction has been re-ignited by schemes that exploit branch correlation. Before 1990, the best branch prediction schemes used the recent history of a branch to predict the future direction of that branch. The most effective dynamic schemes used a table of 2-bit, saturating, up/down counters [26] (often referred to as the branch history table (BHT)), while the most effective static schemes relied on profiles from previous runs of the program [8, 16, 20] to determine a fixed prediction per branch. In 1991, Yeh and Patt [30] introduced two-level adaptive schemes which record the direction of the recently executed branches and use this information (in addition to the branch address) to index into a BHT. This hardware organization allows the prediction scheme to exploit patterns of related branches, increasing the overall prediction accuracy. Pan, So, and Rahmeh [25] appear to have been the first to use the term "correlation." In their two-level adaptive scheme, they use a single hardware shift register of length *k* to record the previous directions of the last *k* branches (Yeh and Patt [31] refer to this scheme as GAs and to *k* as the *history depth*). The contents of the shift register are concatenated with some bits from the branch address to select one of the 2-bit counters in the BHT. Under the constraint of a fixed size BHT, McFarling [18] was able to achieve prediction accuracies better than those from GAs by exclusive-oring (rather than concatenating) bits from the branch address with the bits of the branch history shift register. McFarling refers to this new scheme as *gshare*. Figure 1 illustrates the essential features of GAs and gshare.

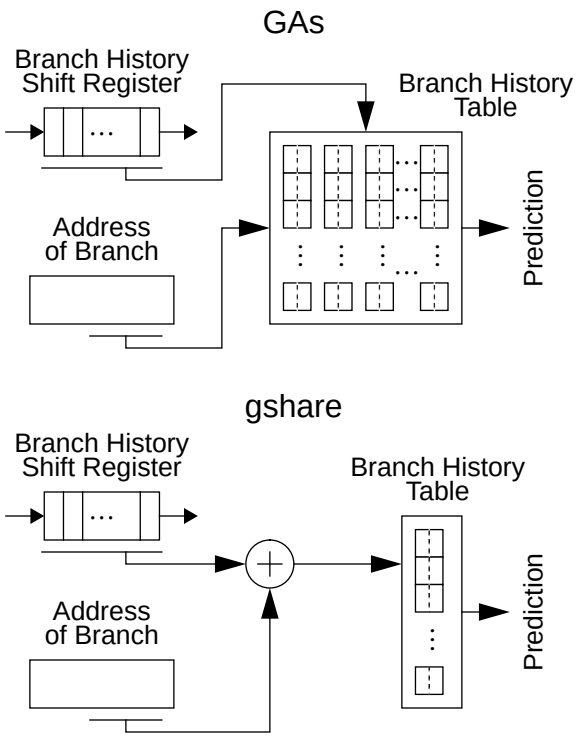

Figure 1. Block diagrams illustrating the hardware organizations for a GAs and a gshare correlated branch prediction scheme. Both use a BHT of 2-bit, up/down, saturating counters.

The results of these hardware studies are appealing because better branch prediction rates translate directly into fewer cycles wasted due to branch mispredictions. However, compile-time optimizations that benefit from improvements in prediction accuracy, such as global instruction scheduling, cannot take advantage of these sophisticated dynamic branch prediction schemes. Inspired by the GAs scheme, Young and Smith [33] developed a static correlated branch prediction (SCBP) scheme that exploits the correlation found in a branch profile to improve overall branch accuracy using only compiler-specified branch prediction bits. SCBP works by encoding branch history into the program counter. As shown in Figure 2, extra copies of blocks are made to differentiate interesting branch histories, i.e. histories that contain branch correlation. Thus, improved prediction accuracy comes at the cost of increased program size.

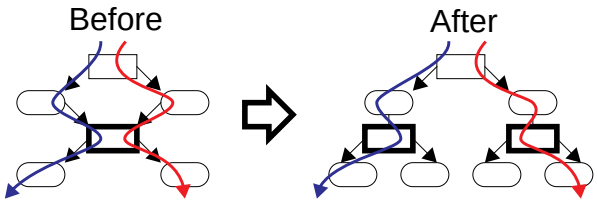

Figure 2. Example illustrating the (simplified) functioning of SCBP. The second IF-block is correlated with the action of the first IF-block. SCBP duplicates the second IF-block so that it can appropriately set the branch prediction bits in the second IF-block.

Young and Smith [33] showed that SCBP does improve overall prediction accuracy over that achievable with simple profiling with reasonable (30-110%) code expansion. They also showed that, by increasing the history depth *k* and thus allowing for greater code expansion, one can achieve even better prediction accuracies. What was beyond the scope of that initial paper was the effect of prediction accuracy and code expansion on performance.

Code expansion during compile-time optimizations is not a new problem. Many compile-time optimizations aimed at exploiting instruction-level parallelism also increase the size of the program text. Loop optimizations, including loop peeling and loop unrolling [23], and software pipelining [5] produce reordered code that is larger than the original. Aggressive function inlining [13] increases the overall code size, even though some savings in code space are realized through the removal of the procedure call overhead and through the enabling of further intra-procedural optimizations. Speculative execution and global instruction scheduling [1, 15, 17, 21, 27] move instructions across basic block boundaries, and this code motion may result in code duplication that expands the size of the program executable. All of these methods increase both the static size and the dynamic memory footprint of the optimized program, placing greater demands on the instruction memory system. Surprisingly, very few of these studies examine the interaction between the code-expanding optimizations and the instruction memory system. Often, studies of these techniques simply assume a perfect instruction memory system and examine only the change in CPU cycle count due to the compile-time optimization (and possibly data cache effects).

A few studies have considered the impact of object code size on instruction memory performance. The earliest of these studies, e.g. Steenkiste [29] and Davidson and Vaughan [7], investigated the relationship between instruction cache performance and code density due to instruction encoding. A later study by Chen et al. [6], fixed the instruction set and examined the impact of code expanding optimizations on the design of instruction caches. They found that several code expanding optimizations noticeably increased the miss ratio of 8 kilobyte and 16 kilobyte caches, and this change resulted in an effective loss of performance after program transformation.

To try and improve the performance of instruction caches, a small number of papers [14, 19, 24] subsequently examined how programs use the instruction memory system and proposed methods to improve its overall performance. Each of the proposed methods uses profiles of previous program runs either to exclude certain portions of the instruction stream from the instruction cache [19] or to reorganize the code layout to avoid conflict misses and improve the spatial and temporal locality of the cache [14, 24]. Since it is often difficult to selectively exclude code from today's instruction caches, we concentrate on the code layout techniques, and in particular on the approach described by Pettis and Hansen [24].[1]

Pettis and Hansen's approach is based on finding an ordering of the procedures of a program such that groups of procedures with frequent calls between them are placed at nearby addresses. They introduce the term "fluff" to refer to code that is not reached during the profiling run or runs. Such code is viewed as error-handling code (or code that handles very rare cases), and they recommend that this fluff be moved to the end of the program in order to compact the part of the program that is actually executed. By compacting the executed part of the program, their approach improves spatial locality and reduces the potential for conflict misses. Pettis and Hansen also describe a method for setting the branch conditions for the taken and

1. The code layout approach by Hwu and Chang [14] is similar, but since it also employs function inlining, it is more appropriate as a compile-time rather than link-time optimization. Our experimental setup modifies object files, and thus link-time optimizations are easier for us to implement than compile-time optimizations.

fall-through paths of branches such that each branch falls through more frequently than it takes. If correctly-predicted taken branches still result in a misfetch penalty (as in the DEC Alpha 21164 [10]), this branch alignment step results in fewer cycles lost due to misfetch penalties and an increase in the average length of straight-line executed code which improves spatial locality. To offset the cache effects of code expansion in SCBP, we have implemented each of these code layout techniques in our experimental system.

Both SCBP and Pettis and Hansen's layout technique rely on good training data. Fisher and Freudenberger [8] found that different data sets are reasonable predictors of other data sets of a program. Needless to say, bad training sets which exercise only small subsets of program features make for bad results.

3 Methodology

We briefly describe the structure of the system that performs our SCBP and code layout transformations, and we discuss the way in which we obtain our measurements. We also outline the influence of the pipeline structure and the instruction cache organization on the performance tradeoffs in branch prediction. Table 1 provides information on the benchmarks that we use in our experiments.

3.1 Experimental system

We use one of two production-quality compilers to generate object files for the DEC Alpha architecture. We then use the ATOM instrumentation tool [28] to generate traces of basic blocks and branch conditions. These traces are needed for both the SCBP and code layout algorithms. Next, we build a procedure call graph and control flow graphs for each procedure using the information in the object files. This step poses some problems for us since it is not always easy to determine the targets of dynamic jumps (i.e. jumps where the target address is computed at runtime), which arise from procedure calls. As a result, the procedure call graph used for the procedure ordering algorithm may not always be complete, and the code layout based on this information may not be optimal in all cases. Finally, the procedure call graph, the control flow graphs, and the profile information are fed to our SCBP and code layout algorithms. After applying SCBP, we use the layout techniques described by Pettis and Hansen [24]. In Section 4, we lump procedure positioning, procedure ordering, procedure placement, and procedure splitting (fluff removal) under the term "code layout", and we refer to basic block placement with the term "branch alignment" (after Calder and Grunwald [2], who solved a similar problem).

Benchmark and Data Set Descriptions		Dynamic Instructions	Dynamic Branches	Static Branches	Original Prog. Size
awk [aw]: pattern-directed scanning/processing, GNU ver. 2.15.5					
a	extensive test of features	18 M	2.5 M	1393	294 KB
c2	analysis of branch profiles	50 M	6.1M	1031	
compress [co]: compression using adaptive Lempel-Ziv, SPECint92					
in	SPECint92 reference input	87 M	11 M	277	63 KB
ps	15-page postscript paper	18 M	2.0 M	268	
diff [di]: differential file comparator, GNU version 2.6					
a	two C files with 3 diffs	5.7 M	432 K	646	174 KB
b	two latex files w/many diffs	3.3 M	275 K	704	
eqntott [eq]: boolean equation to truth table conversion, SPECint92					
fx	8-bit fix to fp encoder	275 M	29 M	533	87 KB
tb	MIPS R2000 branch decode	199 M	19 M	528	
espresso [es]: boolean minimization, SPECint92					
ml	SPECint92 short input	71 M	11 M	1751	247 KB
z5	SPECint92 short input	25 M	3.8 M	1646	
gcc1 [gc]: cc1 program from gcc 2.6.3					
co	compress.c from SPEC92	29M	3.1M	5050	866KB
in	interp.c from SPEC92 sc	36M	3.9M	5144	
grep [gr]: pattern searching program, GNU version 2.0					
re3	search for reg. exp. (21 hits)	2.1 M	325 K	878	135 KB
re5	search for reg. exp. (1K hits)	4.0 M	575 K	827	
sc [sc]: spreadsheet program, SPECint92					
l1	SPECint92 reference input	129 M	23 M	1614	254 KB
l4	modified SPECint92 ref. in.	28 M	5.4 M	1519	
xlisp [li]: lisp interpreter, SPECint92					
n	sqrt() via Newton's method	1.1 M	106 K	550	140 KB
q4	4 queens problem	3.6 M	413 K	605	

Table 1: Benchmark and data set descriptions. The results in this paper were derived from trace-driven simulations. We collected the traces using ATOM v1.1 [28]. We compiled the SPECint92 benchmarks using cc version 2.0.0 and the optimization level specified in the SPEC makefiles. The additional benchmarks were compiled using gcc v2.6.0 (-O3). All of the experiments were performed on a DEC 3000/400 running OSF/1 version 2.0.

The output of the various code transformation algorithms is a set of basic blocks with addresses, static prediction information, and CFG information linking these blocks together. This information allows a trace-driven simulator to generate the statistics on branch and instruction cache behavior that we present in Section 4. We use a simulator instead of running the transformed code on an actual machine for two reasons. From a pragmatic point of view, there are currently very few commercially-available systems that have a processor with static prediction bits. The PowerPC architecture [22] is one of the few incorporating this general functionality, and its most recent processors implement a dynamic branch prediction scheme that takes precedence over the static prediction bits (PowerPC designers believe that dynamic branch prediction schemes perform better than static ones). From an experimental point of view, we want to have the freedom to evaluate performance under several different machine

organizations where we vary only the cache and branch penalties.

3.2 Measuring the influence on performance

To measure the impact of SCBP (and code layout) on performance, we present a metric quantifying the average number of cycles saved per 1000 instructions executed. Unless stated otherwise, the baseline for these numbers is the identical machine microarchitecture under test with profiled branch prediction and no code layout (and hence no code expansion since SCBP was not performed). All of our profile-driven experiments train and test on different inputs.[2] Our performance metric is computed as a weighted sum of the number of mispredicted branches, the number of misfetched branches, and the number of first level (L1) cache misses. The weights in this equation are the branch misprediction and branch misfetch penalties, which are related to the pipeline organization, and the L1 cache miss penalty, which is assumed to be the average amount of time that it takes to fetch the missing block from the rest of the memory system. The larger the ratio of the branch mispredict penalty to the cache miss penalty, the more code expansion the system can tolerate for an improvement in prediction accuracy. For hardware schemes, a larger ratio would shift the optimal balance (all other things being equal) of prediction table size versus cache size in favor of larger prediction tables.

The calculation of our metric assumes that the processor stalls during an instruction cache miss (i.e. the processor does not overlap branch stalls with instruction cache stalls). Even though our metric does not represent overall performance, it is a much better metric than code expansion or even change in instruction cache miss rate. Furthermore, our metric is independent of the rest of the processor organization. It does not matter if the processor issues one instruction per cycle or four instructions per cycle, though obviously, a four-issue machine will benefit more from improvements in prediction accuracy since the cycles saved will be a larger percentage of the total cycles it takes to execute 1000 instructions on a four-issue machine than on a single-issue machine.

For all of the experiments in Section 4, we simulate either an 8 kilobyte or 16 kilobyte direct-mapped instruction cache, each with a 32-byte line size. We chose these design points because the vast majority of high-speed microprocessors include a direct-mapped L1 instruction cache of one of these two sizes. Our results do improve with increasing line size, as expected from the results of the previous papers on code layout, and thus we do not include these simulations in this paper. Since conflict misses occur more often in a direct-mapped rather than a set-associative cache, our results are conservative for an organization with a set-associative L1 instruction cache.

2. For the results in Section 4, we report the result obtained by running on one data set (the testing data set), after having trained on the other (the training data set). The data set listed in the label on the result is the testing data set, e.g. “eq.fx” indicates that the “fx” data set was the testing data set and that “tb” was the training data set for this experiment.

4 Results

To illustrate the combined effects of cache behavior and branch prediction on processor performance, we will present results for three different machine organizations that closely correspond to several recently announced commercial systems. Before we present these performance results however, Sections 4.1 and 4.2 report the effect of SCBP on the code size, the instruction cache miss rate, and the branch misprediction rate. These results provide the background information necessary to understand the performance results presented in Section 4.3.

4.1 Code expansion and cache miss rates

We have measured the code expansion both in terms of the increase of the total program size (Figure 3) and in terms of the increase in the size of the code that is executed during the profiling run (Figure 4). We see that, as the history depth *k* increases, code expansion increases. Since SCBP does not expand those parts of the code that were not executed (i.e. those parts without profile information), the relative increase in the size of the code that is actually fetched into the instruction cache is often much greater than the overall code expansion ratio, especially at large values of *k*. It is this code expansion effect that actually impacts performance.

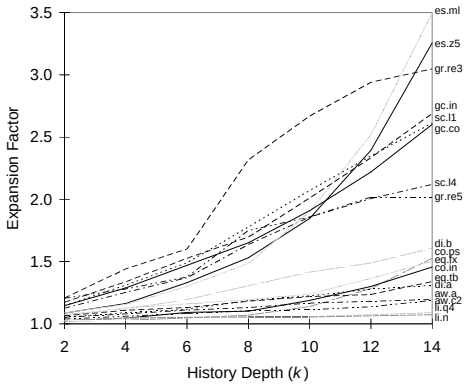

Figure 3. Code expansion of the entire executable due to SCBP. Without SCBP (*k* = 0), the expansion factor is 1.0.

As a first measure of this performance impact, Figure 5 shows the resulting increase in cache misses caused by code expansion in an 8 kilobyte instruction cache. There

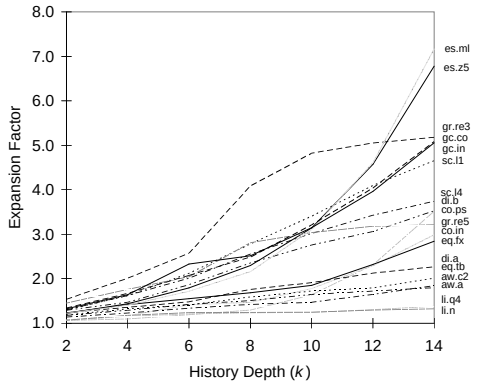

Figure 4. Code expansion due to SCBP in the executed portions of the benchmarks.

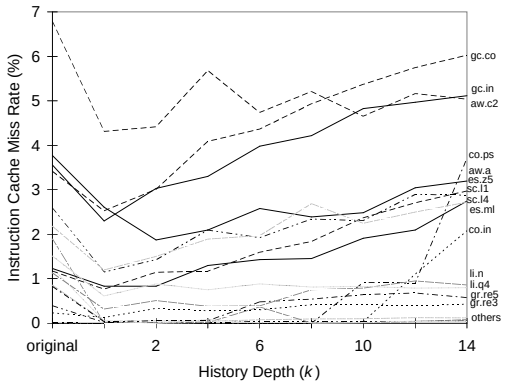

Figure 5. Instruction cache miss rates of untransformed ("original") and transformed ($k \geq 0$) executables. Results are for a direct-mapped instruction cache of 8KB with a 32 byte line. At $k = 0$, we performed code layout and traditional static branch prediction (no SCBP).

are some anomalies in this data that can be explained by conflict misses. Even though we try to remove hot spots from the cache, they still occur occasionally, especially since the behavior of the profiling inputs is different from that of the testing inputs. Overall, the cache miss rate drops after code layout, but then basically increases as *k* increases. The increase, however, is not as dramatic as the code expansion numbers in Figure 4. This effect is due in large part to the code layout routines. Figure 5 demonstrates the significant benefits of code layout via profiling information; often the cache miss rate after code expansion with a large value of *k* is less than the cache miss rate of the original program with procedures in source code order (the first data point in each series of Figure 5).

Finally, Table 2 shows the size of the cache footprint (the number of compulsory misses times the line size) for no layout optimizations and for the endpoint values of *k* with code layout and branch alignment. This table (in combination with Figure 5) shows that, even when the cache footprint is several times larger than our primary instruction cache, code layout is a more important determinant of the cache miss rate (and thus performance) than is the executable size.

Benchmark	Original Program Size	Size after SCBP, code layout, and branch alignment	
		k=0	*k*=14
aw.a	76064	70080	80416
aw.c2	56992	48704	67200
co.in	14944	12352	35392
co.ps	14336	11616	29632
di.a	33504	28160	47936
di.b	35904	30688	50528
eq.fx	29600	24832	40608
eq.tb	29440	24448	43296
es.ml	89504	78656	290400
es.z5	85888	74336	348476
gc.co	297152	268256	571552
gc.in	291456	275808	566560
gr.re3	41024	34560	60384
gr.re5	39456	32608	58560
sc.l	87776	78432	168352
sc.l4	83648	74144	160000
li.n	36096	30912	39136
li.q4	39840	34304	40544

Table 2: Cache footprint of executable in bytes. The cache footprint represents the total number of unique bytes read into the instruction cache (calculated by multiplying the number of compulsory misses times the 32 byte line size).

4.2 Branch prediction accuracy

Figure 6 shows the misprediction rates for two sets of branch prediction schemes: our SCBP scheme, ranging from uncorrelated (*k*=0) to various degrees of correlation (*k*=2 through *k*=14), and the dynamic gshare scheme, with tables ranging in size from 256 bytes (*k*=10) to 8 kilobytes (*k*=15). We chose these prediction table sizes to cover the spectrum of design trade-offs. At 256 bytes, a hardware prediction table is an insignificant hardware cost when compared to the cost of a typical L1 instruction cache. On the other hand, a hardware branch prediction table of 8 kilobytes is the point where an area tradeoff between the branch prediction table and the L1 cache reportedly becomes relevant [12].

Figure 6 illustrates that the prediction accuracy achieved by SCBP is generally not as good as that of gshare. In fact, there are some cases, such as *awk*, *eqntott*, and *xlisp*, where the prediction accuracy is much worse under SCBP. Young et al. [34] discuss a range of reasons why SCBP and gshare achieve different prediction accuracies, and hence, we do not repeat that discussion here. The misprediction rates under SCBP do not monotonically decrease as *k* increases because we train and test on

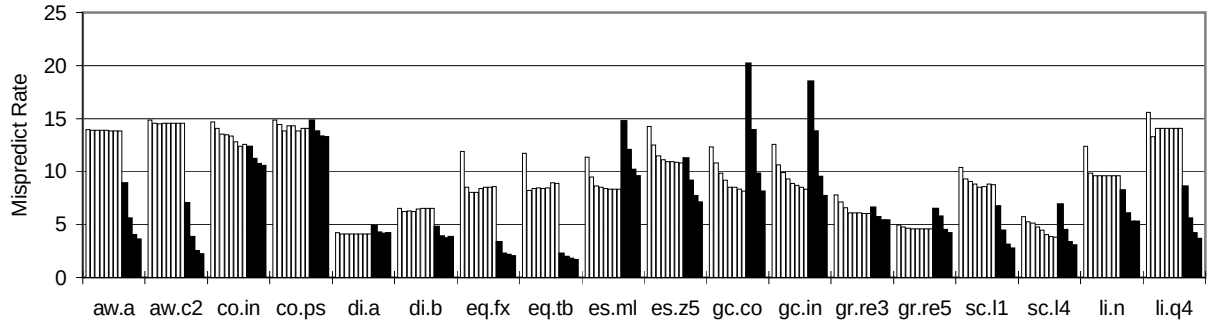

Figure 6. Branch misprediction rates under SCBP and gshare. Each white set of bars depicts the range of values for SCBP with history depths (*k*) ranging from 0 to 14, in steps of 2. Each black set of bars depicts the range of values for gshare with history depth taking values 10, 12, 14, and 15.

different data sets. In fact, the use of more specific information from the training data set (i.e. larger *k* values) can sometimes result in increasingly worse prediction accuracies (e.g. *eq.tb*). For this paper, the important aspect of Figure 6 is that the component of the performance metric due to prediction accuracy will typically be greater under gshare than under SCBP since the prediction accuracy of gshare is typically better than SCBP.

4.3 Evaluating performance

Section 3.2 describes a performance metric, cycles saved per 1000 instructions executed, that focuses on the performance effects of branch prediction, code layout, and code expansion. Through the experiments in the previous subsection, our system can generate the total number of mispredictions saved over a static profiled scheme without branch correlation, the total number of misfetches saved over an executable without branch alignment, and the increase in the number of cache misses over an executable without code expansion and code layout. To evaluate the effects of these changes, we need to choose values for the branch mispredict penalty, the branch misfetch penalty, and the L1 cache miss penalty. Table 3 presents the values that we choose for our simulations. We chose these machine models because their high misprediction penalties demand aggressive branch prediction schemes. Given our limited compile-time use of the branch prediction information, we hypothesized processors with low branch mispredict penalties will not benefit from the few percentage point improvement in prediction accuracy generated by SCBP. Preliminary studies based on a MIPS R2000-like machine, which has less than one cycle of branch misprediction penalty (depending on how the branch delay slot is filled), verified this hypothesis, and so we concentrated our efforts on the next generation of machine models.

Processor model	Mis-predict penalty	Misfetch penalty	Cache miss penalty	Size of L1 Cache
DEC Alpha 21164-like	5	1	6	8KB
Intel P6-like	11	0	3	16KB
HP PA-8000-like	5	2	30	256KB

Table 3: Processor models used in our simulations. The models basically correspond to recently announced microprocessors. We use a L1 instruction cache size of 16KB in our Intel P6-like model since Gwennap [12] reports that Intel could have used a 16KB instruction cache in their P6 processor if they had not implemented a hardware BHT.

Recently announced processors, like the DEC Alpha 21164 [10] and the Intel P6 [11], have implementations more favorable to trading cache misses for mispredictions. The 21164 has a five cycle mispredict penalty and a one cycle misfetch penalty (penalty for correctly predicted taken branches). It incorporates a small 8KB L1 instruction cache and a 96KB on-chip L2 cache with a L1 miss penalty of 6 cycles. The P6 has a branch misprediction penalty of at least 11 cycles and 256KB of on-module, requested-word-first, L2 cache, which reportedly results in a L1 miss penalty of 3 cycles[3]. Other recently announced processors like the HP PA-8000 also benefit from an SCBP scheme since their very large (greater than 256KB) L1 cache is only slightly influenced by the code expansions listed above. Since the cache footprints of almost all of our benchmarks fit completely into a 256KB cache, the few additional cache misses that occur are almost exclu-

3. Our experiments assume that the processor stalls for the entire cache miss penalty. Chen et al. [6] show that the use of requested-word-first miss handling and sequential prefetching can overcome a significant portion of the negative cache effects of code-expanding optimizations. In our P6-like simulations, we assume that the requested-word-first technique hides all but the 3 cycles required to access the L2 cache.

sively compulsory misses due to the differences between the last two columns (labeled *k*=0 and *k*=14) in Table 2.

Table 4 shows all of the detail for our performance calculation using the DEC Alpha 21164-like model in Table 3; Table 5 and Table 6 show the same for the P6-like and PA8000-like machine models. The baseline simulation in Table 4 is a 21164-like machine model with profiled branch prediction (no branch correlation) and no code layout or branch alignment. The numbers in each row correspond to the change in cycles per 1000 instructions due to the component in the row label. Rows labeled "Cache" show the cycles saved (or lost if negative) due to fewer (or more) cache misses. The code layout (procedure ordering and fluff code removal) algorithms lead to cycles saved while code expansion due to the SCBP algorithm potentially lead to cycles lost. Overall, cycles saved due to fewer cache misses typically begins positive at *k*=0 since there is no code expansion and code layout improves the performance of the instruction cache. As *k* increases though, the "Cache" numbers decrease and often become negative at high values of *k*. This trend corresponds to the increasing cost of SCBP's code expansion.

Rows labeled "Predict" show the cycles saved due to fewer branch mispredictions. As expected, the number of cycles saved typically increases as we increase *k*. In benchmarks that exhibit only weak branch correlation (e.g. *diff*), there is very little benefit from SCBP. Furthermore, the benefit of SCBP may fluctuate as *k* increases due to the fact that we train and test on different data sets (the "Prediction" row always improves with increasing *k* when training and testing on the same data set).

Rows labeled "Align" show the benefit due to rearranging the code to make taken branches less frequent. This row contains non-zero numbers only when a machine model, such as the 21164-like model of Table 3, has a non-zero misfetch penalty. In Table 4, branch alignment contributes a large improvement in almost all cases (even though the misfetch penalty is only a single cycle), and the improvement appears to be weakly correlated with *k*. This suggests that the improved predictions under SCBP also improve the effect of branch alignment.

The "Total" row shows the sum of the previous three rows. Overall, the combination of SCBP, layout, and branch alignment often improves performance of the DEC Alpha 21164-like machine model of Table 3. The maximum value in the "Total" row often occurs at a *k* greater than 0, i.e. performance benefits from SCBP. For an 8 kilobyte instruction cache, just the six experiments, *aw.c2*, *es.z5*, *gc.**, and *sc.**, exhibit their best performance tradeoff at *k*=0. The large cache footprints and the large code expansion values for these programs are the main reasons why performance does not improve under SCBP even though the mispredict rates improve with increasing *k*. By enlarging the instruction cache size to 16 kilobytes, the maximum performance benefit occurs at *k*=2 for *aw.c2* and occurs at *k*=4 for *es.z5*. Unfortunately, *gcc1* and *sc* do not benefit from SCBP even at this larger cache size.

Figures 7 through 9 plot three rows from Table 4 that are representative of the types of behavior exhibited by our benchmarks. The three columns per *k* value correspond to the values in each "Cache", "Predict", and "Align" row. The line shows the total of the three components. In general, we see a bell-shaped curve that attains a maximum at some particular value of *k*. In Figure 7, which plots the values for *li.n*, the best performance occurs at *k=6*. In other benchmarks, like *es.ml* (Figure 8), the total line goes negative at high values of *k*. At these high *k* values, the penalty due to code expansion greatly outweighs the benefits of the other components. As mentioned above, a few experiments, like *sc.l1* (Figure 9), perform best at *k*=0.

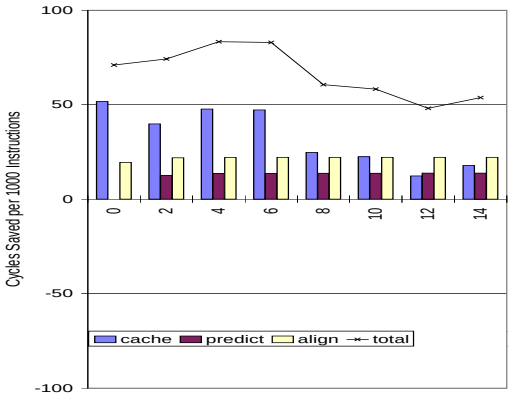

Figure 7. Performance change on the *li.n* benchmark for the 21164-like machine model (8KB cache).

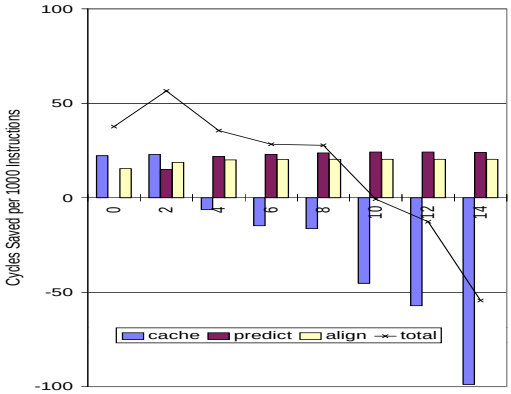

Figure 8. Performance change on the *es.ml* benchmark for the 21164-like machine model (8KB cache).

For the Intel P6-like machine model, we found that the maximum value in the "Total" row occurred at *k* greater than zero for all benchmark runs (Table 5). This result is

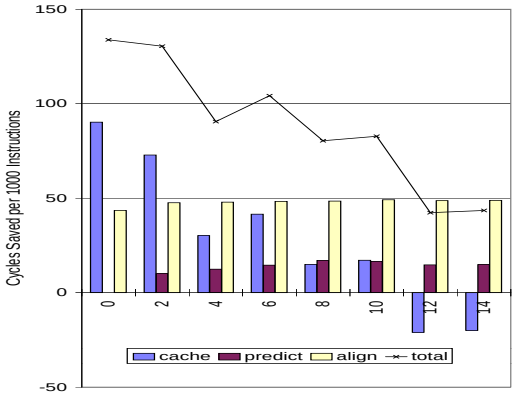

Figure 9. Performance change on the *sc.l1* benchmark for the 21164-like machine model (8KB cache).

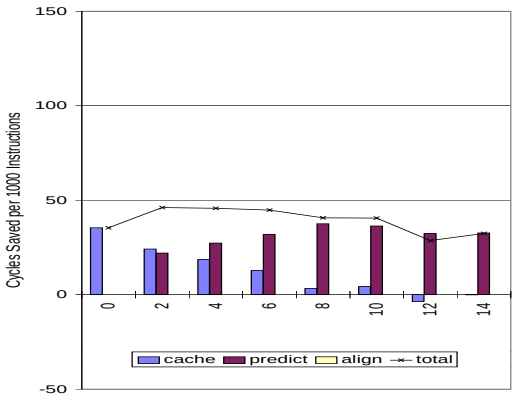

Figure 10. Performance change on the *sc.l1* benchmark for the P6-like machine model (16KB cache). Note that there are no "align" bars because the P6 model does not have a branch misfetch penalty.

not surprising given that the misprediction penalty has increased while the cache miss rates and miss penalties have gotten smaller. Figure 10 re-plots the *sc.l1* results under the P6-like machine model. The total line now peaks at k=2. Also, Figure 10 shows that the magnitude of benefits due to improved branch prediction is comparable to the benefits from code layout in the P6-like model. This trend is a consequence of the bigger ratio of the P6 branch misprediction penalty to cache miss penalty.

In the PA-8000-like simulation, we found that the maximum value in the "Total" row occurred at k greater than zero for all experiments except *diff.**, where the performance is fairly uniform for all values of k. Because of the large misfetch penalty and the fact that our executables can all fit into the L1 cache after code layout, we found that the majority of the benefit in the PA-8000-like simulations comes from the contributions of the code layout techniques and avoided misfetches, rather than from the avoided mispredictions.

4.4 Profiling for performance

It appears that the combination of code layout, SCBP, and branch alignment gives performance benefits at a number of different points in the pipeline and cache design space. Since dynamic correlated branch prediction schemes often achieve better prediction accuracies than SCBP, it is interesting to investigate the performance of a scheme where we replace SCBP by a dynamic correlated branch prediction scheme, such as gshare, and yet retain the benefits of code layout and branch alignment. Figure 11 presents the results of this study for each of our three machine models.

For the 21164-like model, Figure 11 shows that gshare performs significantly better when the executable is first processed by the code layout and branch alignment routines. Without these profile-driven optimizations, the best SCBP scheme (from Table 4) always outperforms gshare. Note that this performance comparison does not penalize gshare for the cost of the 8 kilobyte BHT.

For the P6-like model, Figure 11 shows some, but not much, benefit in gshare when the executable is first processed for code layout and branch alignment. This is a result of the small miss rate (due to a large L1 instruction cache) in the P6-like model. Because code layout is relatively unimportant in the P6-like model, the slightly better branch prediction accuracies under gshare result in noticeably better performance figures than under SCBP.

For the PA-8000-like model, Figure 11 shows that gshare still benefits from code layout, but not quite as much as in the 21164-like model. This is a result of the smaller miss rate in the PA-8000. In all benchmarks except *awk*, the best SCBP scheme outperforms gshare without code-layout optimizations.

The key to the effective use of SCBP is the ability to select the proper value of k. We believe that it is possible to build a compile-time algorithm that is able to select a value of k that is close to the best value of k for each particular application, and thus find a balance between code expansion and prediction accuracy that maximizes performance.

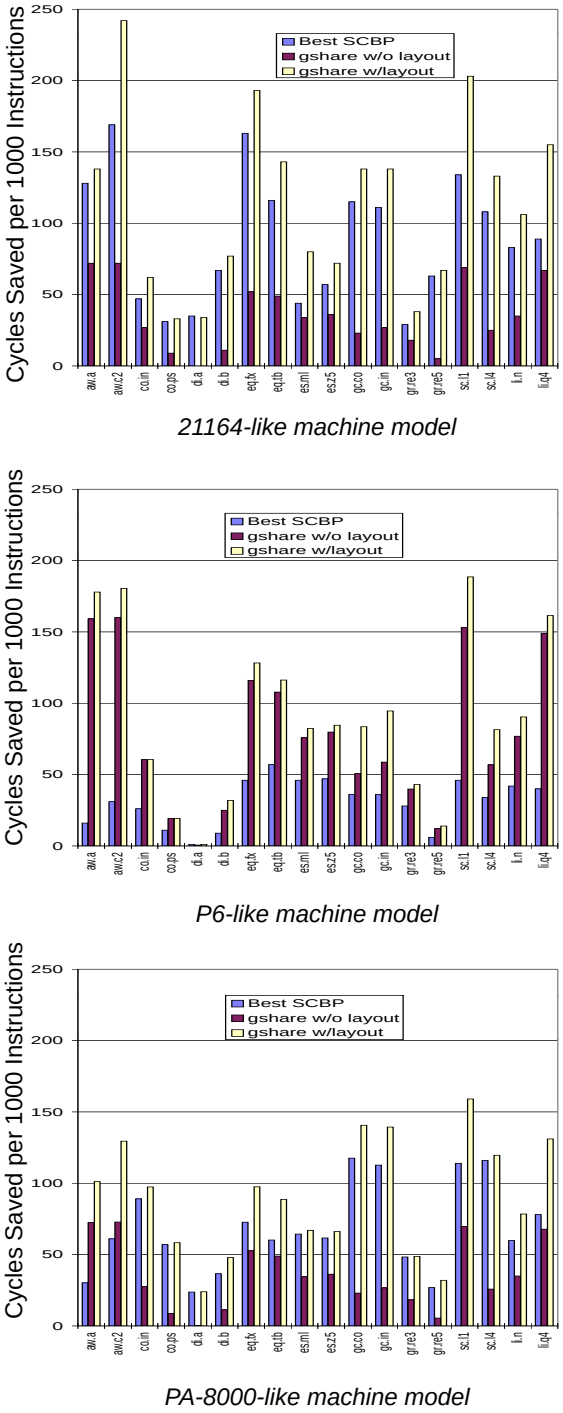

Figure 11. Comparison of performance benefits due to SCBP (including code layout), gshare without layout, and gshare with layout. The contribution due to code layout for gshare is the difference of the “gshare w/o layout” and “gshare w/layout” bars in each group.

5 Conclusion

In this study, we go beyond prediction accuracy to evaluate the performance of SCBP and to quantify the negative effects of code expansion under SCBP. We find that SCBP can improve application performance, especially when coupled with profile-driven code layout and branch alignment techniques. These layout techniques control and minimize the effects of code expansion on the performance of an instruction cache. In fact, we find a synergistic relationship between SCBP and branch alignment in that SCBP also increases the performance improvements resulting from branch alignment. As expected, SCBP achieves the biggest performance gains on machine organizations with high mispredict/misfetch penalties and low cache miss rates/penalties.

In summary, compile-time transformations that maximize prediction accuracy do not necessarily maximize application performance. When small incremental improvements in prediction accuracy result in large amounts of code expansion, there is the potential to improve application performance by limiting the amount of branch correlation exploited by SCBP. To achieve even better performance improvements from incremental changes in prediction accuracy, the next step is to couple SCBP with aggressive ILP techniques like global instruction scheduling, which were not employed in the results of this study.

We also find that a dynamic branch prediction scheme like gshare can benefit significantly from the application of profile-driven code layout and branch alignment techniques. Without the benefit of these profile-driven layout techniques, the performance of gshare may drop markedly. In fact, we find that SCBP with code layout and branch alignment can perform better than gshare without profile-driven layout and alignment. This result is true even when gshare achieves a noticeably lower branch misprediction rate.

6 Acknowledgments

We thank Hewlett-Packard and Digital Equipment Corporation for their generous donation of several HP 9000 Series 700 and DECstation 3000 Series workstations on which we ran our tracing and analysis tools. D. Levitan and M. Surya of IBM Austin provided us with information about PowerPC implementation decisions. Cliff Young is funded by a Graduate Fellowship from the Office of Naval Research. Michael D. Smith is supported by a National Science Foundation Young Investigator award, grant number CCR-9457779.

7 References

[1] D. Bernstein, D. Cohen, and H. Krawcztyk, "Code Duplication: An Assist for Global Instruction Scheduling," *Proc. 24th Annual ACM/IEEE Intl. Symp. and Workshop on Microarchitecture*, Nov. 1991.

[2] B. Calder and D. Grunwald, "Reducing Branch Costs via Branch Alignment", *Proc. 6th Annual Intl. Conf. on Architectural Support for Prog. Lang. and Operating Systems*, Oct. 1994.

[3] P. Chang, E. Hao, T. Yeh, and Y. Patt, "Branch Classification: a New Mechanism for Improving Branch Predictor Performance," in *Proc. 27th Annual ACM/IEEE Intl. Symp. and Workshop on Microarchitecture*, Nov. 1994.

[4] P. Chang, S. Mahlke, and W. Hwu, "Using Profile Information to Assist Classic Compiler Code Optimizations," *Software Practice and Experience*, Vol. 21, No. 12, Dec. 1991.

[5] A. Charlesworth, "An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family," *Computer*, 14(9), Sep. 1981.

[6] W. Chen, P. Chang, T. Conte, and W. Hwu, "The Effect of Code Expanding Optimizations on Instruction Cache Design," Technical Report CRHC-91-17, Coordinated Science Lab, University of Illinois, Urbana, IL, May 1991.

[7] J. Davidson and R. Vaughan, "The Effect of Instruction Set Complexity on Program Size and Memory Performance," *Proc. Second Int. Conf. on Architectural Support for Prog. Lang. and Operating Systems*, Oct. 1987.

[8] J. Fisher and S. Freudenberger, "Predicting Conditional Branch Directions From Previous Runs of a Program," *Proc. 5th Annual Intl. Conf. on Architectural Support for Prog. Lang. and Operating Systems*, Oct. 1992.

[9] L. Gwennap, "Digital Leads Pack with 21164," *Microprocessor Report*, MicroDesign Resources, 8(12), Sep. 12, 1994.

[10] L. Gwennap, "Intel's P6 Uses Decoupled Superscalar Design," *Microprocessor Report*, MicroDesign Resources, 9(2), Feb. 16, 1995.

[11] L Gwennap, "New Algorithm Improves Branch Prediction," *Microprocessor Report*, MicroDesign Resources, 9(4), Mar. 27, 1995.

[12] W. Hwu and P. Chang, "Inlining Function Expansion for Compiling C Programs," *Proc. ACM SIGPLAN 1989 Conf. on Prog. Lang. Design and Implementation*, Jun. 1989.

[13] W. Hwu and P. Chang, "Achieving High Instruction Cache Performance with an Optimizing Compiler," *Proc. of 16th Annual Int. Symp. on Computer Architecture*, May 1989.

[14] W. Hwu, et al., "The Superblock: An Effective Technique for VLIW and Superscalar Compilation", *The Journal of Supercomputing*, Kluwer Academic Publishers, 1993.

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

[16] P. Lowney, S. Freudenberger, T. Karzes, W. Lichtenstein, R. Nix, J. O'Donnell, and J. Ruttenbeg, "The Multiflow Trace Scheduling Compiler," *The Journal of Supercomputing*, Kluwer Academic Publishers, 1993.

[17] S. McFarling, "Combining Branch Predictors," *WRL Technical Note TN-36*, Digital Equipment Corp., Jun. 1993.

[18] S. McFarling, "Program Optimization for Instruction Caches", *Proc. 2nd Annual Intl. Conf. on Architectural Support for Prog. Lang. and Operating Systems*, Oct. 1994.

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

[20] S. Moon and K. Ebcioglu, "An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW Processors," *Proc. 25th Annual ACM/IEEE Intl. Symp. on Microarchitecture*, Dec. 1992

[21] Motorola Corporation, *PowerPC 601 RISC Microprocessor User's Manual*, Motorola, 1993.

[22] D. Padua and M. Wolfe, "Advanced Compiler Optimizations for Supercomputers," *Comm. of the ACM*, 29(12), Dec. 1986.

[23] K. Pettis and R. C. Hansen, "Profile Guided Code Positioning", *Proc. SIGPLAN '90 Conf. on Prog. Lang. Design and Implementation*, Jun. 1990.

[24] 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.

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

[26] M. Smith, "Architectural Support for Compile-Time Speculation," *The Interaction of Compilation Technology and Computer Architecture*, edited by David Lilja and Peter Bird, Kluwer Academic Publishers, 1994.

[27] A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," *Proc. SIGPLAN '94 Conf. on Prog. Lang. Design and Implementation*, Jun. 1994.

[28] P. Steenkiste, "The Impact of Code Density on Instruction Cache Performance," *Proc. 16th Annual Int. Symp. on Computer Architecture*, Jun. 1989.

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

[30] 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.

[31] T. Yeh, "Two-Level Adaptive Branch Prediction and Instruction Fetch Mechanisms for High Performance Superscalar Processors," Computer Science and Engineering Div. Tech. Report CSE-TR-182-93, Univ. of Michigan, Ann Arbor, MI, Oct. 1993.

[32] 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.

[33] C. Young, N. Gloy, and M. Smith, "A Comparative Analysis of Schemes for Correlated Branch Prediction," *Proc. 22nd Annual Intl. Symp. on Computer Architecture*, June1995.

History Depth (k):			0	2	4	6	8	10	12	14
aw	a	Cache	65	113	99	67	80	73	37	27
		Predict	0.0	0.2	0.5	0.5	0.6	0.8	0.8	0.8
		Align	0.0	14	13	13	13	13	0.0	0.0
		Total	65	128	113	81	94	88	37	27
	c2	Cache	142	136	54	115	84	120	87	95
		Predict	0.0	2.0	2.4	2.1	2.0	1.9	1.8	1.8
		Align	27	27	27	28	28	28	28	28
		Total	169	166	84	146	115	151	118	126
co	in	Cache	0.1	0.1	0.1	0.1	0.1	-2.0	-65	-134
		Predict	0.0	4.0	7.4	8.1	8.7	12	15	14
		Align	34	34	35	35	35	36	36	36
		Total	34	38	43	43	43	46	-13	-83
	ps	Cache	0.1	0.1	0.1	0.1	0.1	-58	-57	-238
		Predict	0.0	2.1	5.4	2.8	2.8	5.3	4.0	4.1
		Align	24	24	25	25	25	26	25	25
		Total	24	26	30	28	28	-27	-28	-208
di	a	Cache	22	22	22	22	22	22	21	20
		Predict	0.0	0.4	0.4	0.4	0.4	0.4	0.5	0.5
		Align	11	9.6	9.6	11	11	12	12	12
		Total	34	32	32	34	34	34	34	33
	b	Cache	49	49	49	47	46	46	44	45
		Predict	0.0	1.3	0.9	1.1	0.1	-0.1	-0.2	-0.1
		Align	17	15	15	17	17	17	17	17
		Total	66	65	66	66	64	63	62	62
eq	fx	Cache	118	118	118	95	118	118	116	113
		Predict	0.0	18	20	20	18	18	18	17
		Align	22	25	26	26	26	25	25	25
		Total	141	162	164	142	163	162	160	157
	tb	Cache	74	77	72	77	77	76	74	74
		Predict	0.0	17	16	16	16	15	13	13
		Align	19	21	21	21	21	21	21	21
		Total	94	116	110	114	114	114	109	109
es	z5	Cache	30	6.2	5.2	-22	-37	-72	-94	-111
		Predict	0.0	13	21	24	25	26	26	26
		Align	13	15	16	16	18	18	18	18
		Total	44	34	43	18	6.2	-28	-49	-66
	ml	Cache	22	22	-6.4	-14	-16	-45	-57	-98
		Predict	0.0	15	21	23	23	24	24	24
		Align	15	18	20	20	20	20	20	20
		Total	37	56	35	28	27	-0.7	-12	-54
gc	co	Cache	83	31	-27	-68	-95	-132	-154	-169
		Predict	0.0	8.3	13	17	19	20	21	22
		Align	31	33	34	34	35	35	35	35
		Total	115	72	20	-16	-40	-76	-96	-110
	in	Cache	79	44	-0.4	-68	-77	-109	-121	-124
		Predict	0.0	10	14	18	20	20	22	22
		Align	31	32	33	34	34	34	35	35
		Total	111	88	47	-16	-22	-53	-63	-66
gr	re3	Cache	6.6	-6.3	-2.9	-4.1	-11	-11	-10	-12
		Predict	0.0	4.7	9.0	12	12	12	13	13
		Align	13	18	19	19	19	19	19	19
		Total	20	16	25	28	21	20	22	20
	re5	Cache	49	48	48	22	18	11	9.7	16
		Predict	0.0	1.3	2.1	2.6	2.5	2.5	2.5	2.5
		Align	12	12	13	13	13	13	13	13
		Total	61	62	63	37	33	27	25	31
sc	l1	Cache	90	72	30	41	14	17	-21	-20
		Predict	0.0	10	12	14	17	16	14	14
		Align	43	47	47	48	48	49	48	48
		Total	133	130	90	104	80	82	42	43
	l4	Cache	62	43	19	14	-30	-1.9	-17	-30
		Predict	0.0	4.5	5.9	9.3	12	16	17	18
		Align	45	50	51	51	50	50	50	50
		Total	107	98	76	74	32	65	51	38
li	n	Cache	51	39	47	47	24	22	12	17
		Predict	0.0	12	13	13	13	13	13	13
		Align	19	21	22	22	22	22	22	22
		Total	71	74	83	82	60	58	48	53
	q4	Cache	56	39	48	39	44	43	45	45
		Predict	0.0	13	8.5	8.5	8.5	8.5	8.5	8.5
		Align	30	32	32	32	32	32	32	32
		Total	87	84	88	80	84	83	86	85

Table 4: Cycles saved per 1000 instructions under SCBP and code layout. The machine model assumed is the **DEC Alpha 21164-like processor with 8KB** of direct-mapped instruction cache (32 byte line size). The base line is the same machine model with profiled prediction (k=0) and no code layout or branch alignment.

History Depth (k):			0	2	4	6	8	10	12	14
aw	a	Cache	18	16	23	6.1	8.4	3.0	3.6	0.1
		Predict	0.0	0.4	1.1	1.1	1.3	1.8	1.7	1.7
		Align	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
		Total	18	17	24	7.2	9.6	4.8	5.3	1.9
	c2	Cache	20	31	13	27	17	20	-3.9	3.4
		Predict	0.0	4.4	5.2	4.7	4.4	4.3	4.0	4.0
		Align	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
		Total	20	35	18	32	22	25	0.1	7.4
co	in	Cache	0.0	0.0	0.0	0.0	0.0	-0.5	-2.0	-33
		Predict	0.0	8.8	16	17	19	26	33	30
		Align	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
		Total	0.0	8.9	16	18	19	26	31	-2.2
	ps	Cache	0.0	0.0	0.0	0.0	0.0	0.0	-0.4	-24
		Predict	0.0	4.6	11	6.1	6.1	11	8.8	8.9
		Align	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
		Total	0.0	4.7	11	6.2	6.2	11	8.4	-15
di	a	Cache	0.4	0.3	0.2	0.1	0.1	0.0	0.0	-0.1
		Predict	0.0	0.9	0.9	0.8	0.8	0.8	1.1	1.1
		Align	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
		Total	0.4	1.2	1.2	0.9	0.9	0.8	1.1	1.0
	b	Cache	6.9	6.5	6.7	6.4	6.3	6.2	6.1	6.0
		Predict	0.0	2.8	1.9	2.5	0.3	-0.3	-0.4	-0.3
		Align	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
		Total	6.9	9.4	8.6	8.9	6.5	5.9	5.7	5.7
eq	tb	Cache	8.4	8.4	6.1	8.4	8.4	8.4	7.7	8.0
		Predict	0.0	37	35	35	35	35	30	30
		Align	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
		Total	8.4	46	41	43	43	43	37	38
	fx	Cache	12	12	11	0.8	12	12	12	12
		Predict	0.0	40	45	45	40	39	39	39
		Align	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
		Total	12	52	57	45	53	52	51	51
es	z5	Cache	6.5	4.1	-2.0	-5.2	-14	-15	-26	-41
		Predict	0.0	33	48	50	52	53	53	52
		Align	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
		Total	6.5	37	46	45	37	37	26	11
	ml	Cache	4.8	0.7	0.3	-12	-10	-28	-34	-46
		Predict	0.0	29	47	53	56	57	57	58
		Align	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
		Total	4.8	30	47	41	45	28	23	11
gc	co	Cache	32	18	-15	-23	-41	-54	-66	-74
		Predict	0.0	18	29	38	42	45	48	50
		Align	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
		Total	32	36	14	14	1.8	-9.1	-18	-24
	in	Cache	35	13	5.1	-15	-23	-42	-46	-50
		Predict	0.0	23	31	39	43	46	48	50
		Align	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
		Total	35	36	36	23	20	4.0	2.3	-0.4
gr	re3	Cache	1.7	-3.3	-2.7	-2.4	-5.6	-5.0	-4.5	-4.6
		Predict	0.0	10	19	28	27	27	28	28
		Align	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
		Total	1.7	7.1	17	26	22	22	24	24
	re5	Cache	1.8	1.4	1.3	1.2	0.8	0.8	0.6	0.7
		Predict	0.0	2.9	4.5	5.6	5.5	5.6	5.6	5.6
		Align	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
		Total	1.8	4.2	5.8	6.8	6.3	6.3	6.2	6.2
sc	l1	Cache	35	24	18	12	3.2	4.2	-3.7	-0.2
		Predict	0.0	22	27	31	37	36	32	32
		Align	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
		Total	35	46	45	44	40	40	28	32
	l4	Cache	24	16	7.8	2.7	-16	-0.9	-9.2	-11
		Predict	0.0	9.8	12	20	26	35	39	41
		Align	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
		Total	24	26	20	23	10	34	30	29
li	n	Cache	13	11	12	12	12	12	11	10
		Predict	0.0	27	29	30	30	30	30	30
		Align	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
		Total	13	39	42	42	42	42	41	40
	q4	Cache	12	5.9	8.3	5.1	7.2	6.6	19	21
		Predict	0.0	28	18	18	18	18	18	18
		Align	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
		Total	12	34	27	23	26	25	38	40

Table 5: Cycles saved per 1000 instructions under SCBP and code layout. The machine model assumed is the **Intel P6-like processor with 16KB** of direct-mapped instruction cache (32 byte line size). The base line is the same machine model with profiled prediction (k=0) and no code layout or branch alignment.

History Depth (*k*):			0	2	4	6	8	10	12	14
aw	a	Cache	0.39	0.25	0.05	-0.10	-0.22	-0.35	-0.49	-0.43
		Predict	0.00	0.18	0.49	0.50	0.58	0.83	0.78	0.79
		Align	28	29	28	28	28	28	28	28
		Total	28	30	28	28	28	28	28	28
	c2	Cache	0.16	0.11	0.03	-0.03	-0.08	-15	-27	-0.19
		Predict	0.00	2.00	2.36	2.14	2.00	1.94	1.82	1.82
		Align	56	57	57	58	58	58	58	58
		Total	56	59	60	60	60	45	33	60
co	in	Cache	0.03	0.03	0.03	0.02	0.00	-0.04	-0.11	-0.22
		Predict	0.00	4.01	7.44	8.15	8.70	12	15	13
		Align	69	69	71	71	70	72	74	73
		Total	69	73	78	79	79	84	89	87
	ps	Cache	0.14	0.13	0.10	0.00	-0.02	-0.20	-0.43	-0.76
		Predict	0.00	2.09	5.36	2.78	2.78	5.29	4.01	4.07
		Align	49	49	50	51	51	51	51	51
		Total	49	51	56	53	53	57	54	54
di	a	Cache	0.95	0.27	-0.06	-0.68	-1.20	-1.73	-2.01	-2.28
		Predict	0.00	0.40	0.42	0.36	0.36	0.37	0.51	0.50
		Align	22	20	20	23	23	24	24	24
		Total	23	21	20	23	22	23	23	22
	b	Cache	1.58	0.33	-0.45	-1.18	-2.40	-3.03	-3.76	-4.08
		Predict	0.00	1.29	0.86	1.12	0.12	-0.14	-0.18	-0.15
		Align	35	32	32	36	36	36	36	36
		Total	36	34	33	36	33	32	32	31
eq	tb	Cache	0.02	0.01	-0.01	-0.01	-0.02	-0.03	-0.06	-0.07
		Predict	0.00	17	16	16	16	15	13	13
		Align	39	43	43	43	43	43	42	42
		Total	39	60	59	59	59	59	56	56
	fx	Cache	0.02	0.01	0.00	-0.01	-0.02	-0.02	-0.03	-0.04
		Predict	0.00	18	20	20	18	18	18	17
		Align	44	51	52	52	52	51	51	51
		Total	44	69	72	72	70	69	69	69
es	z5	Cache	0.13	-0.04	-0.19	-0.46	-0.80	-1.31	-5.85	-27
		Predict	0.00	15	21	23	23	24	24	24
		Align	32	38	40	41	41	41	41	41
		Total	32	53	62	63	64	64	59	38
	ml	Cache	0.40	-0.07	-0.43	-1.02	-1.76	-3.15	-16	-32
		Predict	0.00	13	21	24	25	25	26	26
		Align	29	32	34	34	37	37	38	38
		Total	29	45	55	58	61	60	48	32
gc	co	Cache	53	22	15	-88	-39	-135	-91	-115
		Predict	0.00	8.33	13	17	19	20	21	22
		Align	64	66	68	70	70	70	71	71
		Total	117	97	97	-1.04	50	-44	1.67	-20
	in	Cache	49	31	-21	-32	-72	-72	-127	-121
		Predict	0.00	10	14	17	19	20	22	22
		Align	63	66	67	69	69	69	70	70
		Total	112	108	61	54	17	18	-34	-28
gr	re3	Cache	2.87	-1.15	-3.01	-4.25	-6.72	-7.49	-8.25	-8.42
		Predict	0.00	4.73	9.00	12	12	12	13	12
		Align	27	36	38	39	39	39	39	39
		Total	30	40	44	48	45	44	44	44
	re5	Cache	1.56	-0.04	-1.16	-1.75	-2.74	-3.70	-4.13	-4.47
		Predict	0.00	1.31	2.06	2.55	2.50	2.53	2.54	2.53
		Align	24	25	25	26	25	25	25	25
		Total	26	26	26	26	25	24	24	24
sc	l1	Cache	0.06	-0.04	-0.11	-0.26	-1.35	-2.02	-2.01	-4.64
		Predict	0.00	9.99	12	14	17	16	14	14
		Align	89	96	97	98	98	99	98	98
		Total	89	106	109	112	113	113	111	108
	l4	Cache	0.30	-0.18	-0.46	-0.88	-1.43	-3.05	-7.58	-9.89
		Predict	0.00	4.45	5.85	9.30	12	16	17	18
		Align	93	102	103	103	102	102	102	103
		Total	93	106	108	111	113	116	112	111
li	n	Cache	4.57	3.15	2.06	1.09	0.61	0.36	-1.09	-2.59
		Predict	0.00	12	13	13	13	13	13	13
		Align	38	43	44	44	44	44	44	44
		Total	43	59	59	58	58	58	56	55
	q4	Cache	1.46	1.04	0.71	0.43	0.31	0.38	0.05	-0.15
		Predict	0.00	12	8.50	8.48	8.53	8.49	8.51	8.53
		Align	61	64	64	64	64	64	64	64
		Total	63	78	73	73	73	73	72	72

Table 6: Cycles saved per 1000 instructions under SCBP and code layout. The machine model assumed is the **HP PA8000-like processor with 256KB** of direct-mapped L1 instruction cache (32 byte line size). The base line is the same machine model with profiled prediction (*k*=0) and no code layout or branch alignment.

Benchmark	History depth k						
	2	4	6	8	10	12	14
awk.a	1.052	1.080	1.110	1.131	1.166	1.182	1.196
awk.c2	1.040	1.060	1.082	1.104	1.113	1.137	1.189
com.in	1.025	1.045	1.092	1.101	1.188	1.300	1.457
com.ps	1.022	1.028	1.047	1.069	1.139	1.283	1.525
diff.a	1.052	1.091	1.117	1.186	1.226	1.279	1.310
diff.b	1.073	1.119	1.195	1.305	1.415	1.488	1.608
eqn.tbra	1.065	1.108	1.131	1.182	1.218	1.235	1.340
eqn.fx2fp	1.070	1.128	1.159	1.195	1.239	1.365	1.502
esp.mlp4	1.089	1.156	1.300	1.485	1.900	2.518	3.499
esp.z5	1.087	1.163	1.330	1.531	1.847	2.394	3.255
grep.re3	1.206	1.441	1.600	2.318	2.669	2.941	3.047
grep.re5	1.205	1.279	1.376	1.749	1.858	2.018	2.014
sc.load1	1.146	1.311	1.489	1.178	2.072	2.348	2.613
sc.load4	1.125	1.251	1.369	1.638	1.856	2.005	2.119
li.new	1.023	1.046	1.053	1.054	1.055	1.064	1.073
li.qu4	1.024	1.045	1.053	1.056	1.058	1.074	1.090

Table 7: Code expansion of the entire executable due to SCBP. Figure 3 was created from this data.

Benchmark	History Depth k						
	2	4	6	8	10	12	14
awk.a	1.177	1.291	1.425	1.507	1.645	1.721	1.797
awk.c2	1.136	1.227	1.332	1.421	1.472	1.646	1.844
com.in	1.077	1.167	1.413	1.464	1.777	2.292	2.992
com.ps	1.063	1.093	1.188	1.296	1.635	2.310	3.519
diff.a	1.237	1.354	1.481	1.759	1.910	2.125	2.269
diff.b	1.290	1.469	1.865	2.352	2.755	3.059	3.528
eqn.tbra	1.186	1.323	1.409	1.588	1.724	1.791	2.010
eqn.fx2fp	1.229	1.420	1.552	1.685	1.848	2.321	2.841
esp.mlp4	1.222	1.402	1.717	2.157	3.106	4.607	7.170
esp.z5	1.219	1.435	1.791	2.298	3.143	4.569	6.779
grep.re3	1.535	2.011	2.572	4.087	4.822	5.053	5.180
grep.re5	1.452	1.759	2.047	2.808	3.038	3.182	3.214
sc.load1	1.298	1.616	2.122	2.783	3.409	4.094	4.650
sc.load4	1.297	1.624	2.010	2.555	3.020	3.429	3.742
li.new	1.066	1.174	1.236	1.243	1.248	1.295	1.325
li.qu4	1.063	1.155	1.211	1.231	1.241	1.305	1.382

Table 8: Code expansion of the executed portions of the benchmarks due to SCBP. Figure 4 was created from this data.

Benchmark	Original program	SCBP with profiled code layout and branch alignment							
		k=0	2	4	6	8	10	12	k=14
awk.a	3.770	2.608	1.870	2.092	2.578	2.387	2.485	3.043	3.196
awk.c2	6.786	4.311	4.416	5.683	4.738	5.212	4.652	5.160	5.038
com.in	0.001	0.001	0.001	0.001	0.001	0.001	0.032	1.019	2.079
com.ps	0.003	0.002	0.002	0.002	0.002	0.002	0.918	0.901	3.708
diff.a	0.396	0.020	0.024	0.025	0.029	0.030	0.034	0.037	0.054
diff.b	0.860	0.047	0.054	0.044	0.077	0.093	0.100	0.125	0.117
eqn.fx2fp	1.894	0.001	0.002	0.014	0.368	0.005	0.009	0.031	0.081
eqn.tbra	1.237	0.042	0.001	0.074	0.004	0.006	0.016	0.047	0.047
esp.mlp4	1.231	0.766	1.146	1.169	1.598	1.836	2.369	2.707	2.972
esp.z5	1.174	0.831	0.833	1.296	1.430	1.454	1.907	2.093	2.743
grep.re3	0.232	0.127	0.332	0.279	0.299	0.413	0.417	0.395	0.426
grep.re5	0.833	0.036	0.051	0.054	0.479	0.542	0.641	0.677	0.573
sc.load1	2.587	1.150	1.430	2.092	1.923	2.341	2.307	2.896	2.880
sc.load4	2.184	1.195	1.499	1.890	1.973	2.689	2.246	2.497	2.719
li.new	1.133	0.317	0.509	0.382	0.390	0.750	0.787	0.949	0.861
li.qu4	1.517	0.614	0.885	0.748	0.881	0.813	0.825	0.788	0.798

Table 9: Instruction cache miss rates of untransformed and transformed (code layout and SCBP) executables. Results are for a direct-mapped instruction cache of 8KB with a 32 byte line. Figure 5 is based on this data.

Benchmark	Original program	SCBP with profiled code layout and branch alignment							
		k=0	2	4	6	8	10	12	k=14
awk.a	1.498	0.865	0.917	0.714	1.247	1.179	1.343	1.326	1.431
awk.c2	3.065	2.311	1.974	2.524	2.093	2.389	2.304	3.060	2.836
com.in	0.003	0.001	0.001	0.001	0.001	0.001	0.015	0.064	1.023
com.ps	0.001	0.002	0.002	0.002	0.002	0.002	0.003	0.015	0.750
diff.a	0.029	0.016	0.020	0.021	0.025	0.026	0.029	0.030	0.032
diff.b	0.259	0.030	0.044	0.039	0.048	0.053	0.055	0.057	0.061
eqn.fx2fp	0.268	0.001	0.001	0.013	0.367	0.002	0.001	0.002	0.002
eqn.tbra	0.389	0.001	0.001	0.073	0.001	0.001	0.002	0.022	0.013
esp.mlp4	0.493	0.308	0.436	0.453	0.845	0.800	1.353	1.531	1.902
esp.z5	0.455	0.293	0.371	0.565	0.665	0.954	0.989	1.327	1.802
grep.re3	0.113	0.059	0.217	0.199	0.191	0.294	0.275	0.258	0.260
grep.re5	0.086	0.028	0.042	0.045	0.047	0.061	0.061	0.066	0.064
sc.load1	1.524	0.407	0.761	0.935	1.115	1.417	1.384	1.627	1.518
sc.load4	1.402	0.621	0.890	1.161	1.325	1.941	1.451	1.721	1.794
li.new	0.562	0.135	0.199	0.175	0.180	0.177	0.179	0.199	0.228
li.qu4	0.775	0.377	0.586	0.507	0.609	0.541	0.561	0.166	0.095

Table 10: Instruction cache miss rates of untransformed and transformed (code layout and SCBP) executables. Results are for a direct-mapped instruction cache of 16KB with a 32 byte line.