Novel Techniques to Improve Branch Prediction Accuracy for Embedded Processors in the Presence of Context Switches

Sudeep Pasricha, Alex Veidenbaum
{sudeep, alexv} @cecs.uci.edu

CECS Technical Report 03-24
August 2003

Center for Embedded Computer Systems
School of Information and Computer Science
University of California, Irvine,
Irvine, CA 92697-3425, USA

Abstract

Embedded processors like Intel's XScale use dynamic branch prediction to improve performance. Due to the presence of context switches, the accuracy of these predictors is reduced because they end up storing prediction histories for several processes. This paper shows that the loss in accuracy can be significant and depends on predictor type and size. Several new schemes are proposed to save and restore the predictor state on context switches in order to improve prediction accuracy. The schemes differ in the amount of information they save and vary in their accuracy improvement. It is shown that even for a small 1K entry hybrid predictor, 1 - 6% improvement in prediction rate can be achieved (for an average context switch interval of 100K instructions) for several embedded applications while saving and restoring a minimal amount of state information (less than 128 bits) on a context switch.

1. Introduction

Modern embedded processors use pipelining to exploit parallelism and improve performance. Conditional branches in the instruction stream degrade performance by causing pipeline flushes. Branch prediction mechanisms can overcome this limitation by predicting the outcome of the branch before its condition is resolved. As a result, instruction fetch is not interrupted as often and the window of instructions over which ILP can be exposed increases. In fact, accurate branch predictors can eliminate over 90% [13] of these pipeline stalls and are thus critical to realizing the performance potential of a processor. Improving branch prediction accuracy is important because the new generation of embedded processors have deeper pipelines, which result in larger misprediction penalties. In the XScale [20] processor which has a 7 stage pipeline, the penalty for each misprediction is as much as 4 cycles.

Most processors use dynamic branch prediction [5, 6, 11-13, 15] to predict branch directions. Dynamic predictors record and utilize information from previous runs of a static branch instruction to predict its outcome in the future. This requires additional hardware to store the branch history. These predictors dynamically adjust their prediction to match the changing behavior of a branch instruction as the program executes.

One aspect of branch prediction that has largely been ignored is the effect of context switches. In typical systems, several processes are in the active queue at any given time and they share the branch predictor structure. Each process runs for its allotted time slice and then yields the processor to allow another waiting process to execute. Unless steps are taken to change the state of the predictor structure, it will contain stale information from the run of the previous process when the new process commences execution. Since different processes generally have completely different branch behaviors, reusing the stale information will increase the misprediction rate. It is desirable to overcome this limitation. This research explores the effects of context switches on dynamic branch prediction schemes, in the context of embedded processors which have stringent hardware resource constraints. We find that context switches cause substantial decrease in prediction rate – as high as 10% for the hybrid predictor for some cases! Having established that context switches degrade prediction accuracy, we then propose several methods to alleviate this performance loss for the hybrid scheme, at different costs to the architect. We choose the hybrid predictor because it is widely used, but the proposed schemes work for other dynamic predictors too.

This paper is organized as follows: Section 2 presents previous work in this area. Section 3 describes various dynamic branch prediction schemes that we have considered for our experiments. Section 4 illustrates the effect of context switches on these dynamic branch predictors. Section 5 presents schemes to improve branch predictor performance in the presence of context switches. Section 6 reports the simulation results and our analysis for the proposed schemes. Section 7 provides some concluding remarks.

2. Related Work

Several papers on branch prediction acknowledge the effects of context switching on branch prediction accuracy.

Yeh and Patt [6] examined the effect of context switches on two-level branch prediction schemes. They found that the average accuracy degradations for the PAp, PAg and GAg schemes are less than one percent for a context switch interval of 500K instructions. However in their experiments they did not change the pattern history table on a context switch, which explains the exceptionally small decrease in prediction accuracy for the large predictor structures used. In an actual multi-programming environment, the pattern history tables for different processes will differ and if the PHT is kept unchanged, prediction accuracy will suffer.

Gloy, et al [7] studied dynamic branch prediction schemes on system workloads. They found that including kernel level branches with user level branches in experiments significantly affected branch prediction accuracies, increasing aliasing and thus decreasing prediction accuracy. They emphasized the need to consider the whole system rather than just user level code when evaluating branch prediction schemes. This motivated us to study the impact of context switches on the performance of dynamic prediction schemes.

More recently, Michele Co. and K. Skadron [1] claimed that context switching has negligible effect on branch predictor performance. They measured the context switch interval based on the default time slice value for Windows NT (25 ms), and it turned out to be around 50M instructions. Our experiments calculate this interval from context switching information obtained from several multi-programmed systems with varying workloads. The interval we obtain is much smaller than theirs and agrees with the findings of previous studies [2, 3, 8, 10] that consider the effect of context switching on branch prediction.

A.S. Dhodapkar and James E. Smith [3] presented the case for saving predictor information on a context switch. They proposed (for a gshare predictor) saving the most significant bits of all the counters in the branch predictor tables on a context switch. They also proposed setting the predictor counters to weakly taken on a context switch, as an alternative. This work is the only one that we know of which proposes mechanisms to

improve predictor accuracy in the presence of context switches. They rightly identify the need to reduce the 'learning time' of the predictor after a context switch by restoring previously saved prediction values into the predictor. This paper extends this study and proposes several other mechanisms that will improve performance.

3. Dynamic Branch Prediction Schemes

Dynamic branch prediction has proven to be an extremely powerful technique for accurately predicting branch direction. Several schemes have been proposed [5, 6, 11-13, 15] which exploit different branch characteristics to better predict their outcome. In the remainder of this section we look at five such schemes which we have considered in our experiments to determine the effect of context switches on branch prediction.

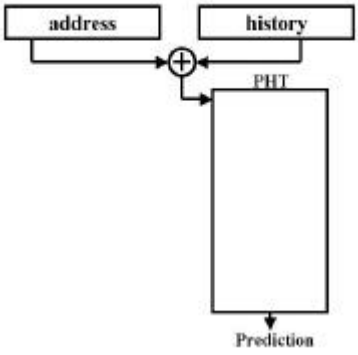


Figure 1. Gshare predictor

The gshare [12] scheme (Figure 1) utilizes the global history of a branch, which works well because branches tend to be highly correlated in applications. A global history register of size *n* bits stores the outcomes of the last *n* branch instructions executed by the program.

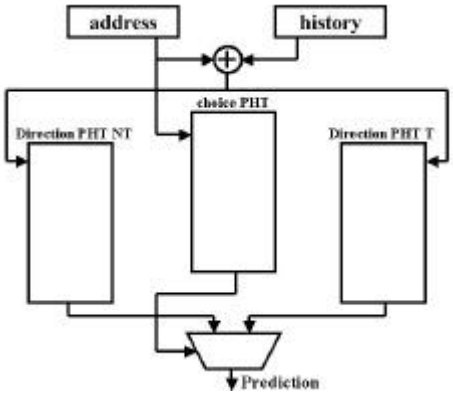


Figure 2. Bimode predictor

This register is xor'ed with the branch address to access a table of two bit saturating counters (referred to as a pattern history table or PHT), which gives the direction prediction. The xor'ing reduces destructive aliasing by better distributing different branches to separate locations in the table.

The bi-mode predictor [13] tries to reduce aliasing by separating the "taken" branches from the "not taken" (Figure 2). It has a two-bit counter table for each of these, called the direction PHT, and another table called the choice PHT to select between these two tables. The branch address is used to access the choice PHT, while the direction PHTs are accessed by xor'ing global history to the branch address. The choice PHT selects from the "taken" and the "not taken" direction PHTs to provide the prediction. The updating mechanism ensures that branches that are biased to be taken will have their predictions in the "taken" direction PHT, while branches biased towards not being taken will have their predictions in the "not taken" branch PHT.

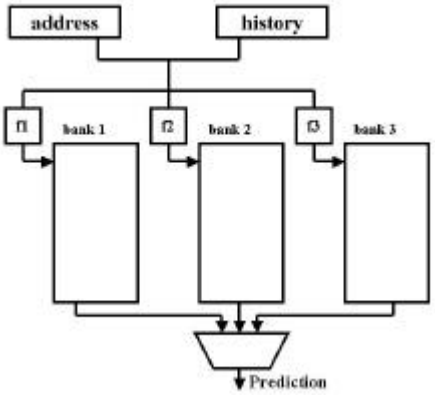


Figure 3. Skewed predictor

The skewed branch predictor [15] attempts to reduce aliasing by increasing associativity in the PHTs. Instead of using tags which would not be cost-effective, it emulates associativity by using skewing functions to access PHT locations (Figure 3). The predictor consists of three PHTs, each accessed by a unique hashing function. The tables are simple arrays of two bit saturating counters. The prediction is made according to a majority vote among the three prediction values. Partial updating of these tables further reduces aliasing and improves prediction accuracy.

The hybrid predictor [12] combines multiple prediction strategies into a single predictor. A selection mechanism is used to determine the most suitable component predictor for predicting each branch. For two component hybrid predictors, a selection table of two bit saturating counters is used (Figure 4). The most common hybrid configuration combines a bimodal predictor (a

table of two bit saturating counters) with a gshare predictor.

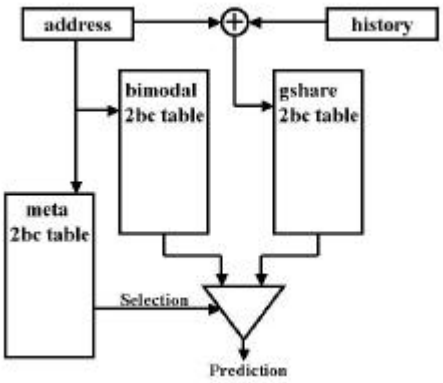


Figure 4. Hybrid predictor

The alloy predictor [11] combines global and local history when predicting the direction of a branch (Figure 5). Bits selected from the branch address, the global history register and the local history table (the branch history table or BHT in Figure 5) are concatenated together to access a two-bit counter in the PHT and obtain a prediction. The authors claim that this scheme reduces PHT aliasing because branches that alias with one type of history are often distinguished by the other type of history.

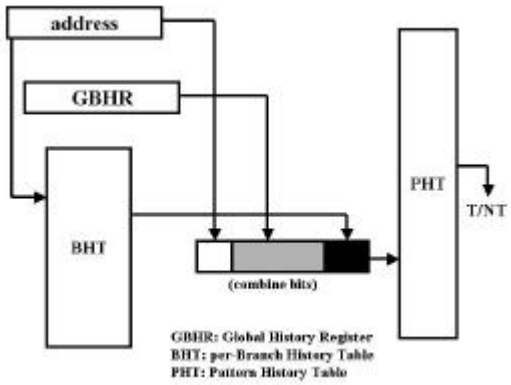


Figure 5. Alloy predictor

4. Effect of Context Switches

Context switches can occur during program execution for several reasons such as I/O requests, system calls, page faults, expiration of time slice etc. The frequency of these context switches depends on factors like the number of applications active on a system, the types of these applications, the operating system used and the scheduling scheme. We performed experiments to determine context switch intervals on several different systems, with varying workloads and running different operating systems such as UNIX, Linux and Windows 2K. For UNIX and Linux we used the *vmstat* utility while for Windows we used the *ntimer* utility, which is part of the Windows 2000 Resource Kit. Our results indicate a context switch frequency varying from 100/sec to 8000/sec and a context switch interval ranging from 75K – 1000K instructions. For instance, one of the machines we tested was a SUN UltraSparc-II workstation running SunOS 5.8 at a maximum clock speed of 400 MHz. The context switch frequency on it varied from 400/sec to 4500/sec, with a changing workload. If it is assumed that one instruction is executed every cycle, we get a context switch interval of 90K instructions for the higher end. In another experiment, we tested a 996 MHz Intel PIII machine running Windows 2000 and found that the context switch frequency varied from about 1000/sec to 8000/sec which gives a context switch interval of around 125K instructions for the higher end, assuming an IPC of 1. These numbers are in line with the results obtained in [3] as well as other studies done previously that analyze branch predictor performance in the presence of context switches [2, 8, 10].

Figure 6 shows the average performance degradation for several commonly used dynamic branch prediction schemes in the presence of context switches, for 25 benchmarks from the MiBench [21] suite. “no CS” represents the ideal case when no context switches occur. We do not modify the branch predictor tables on a context switch, allowing the prediction information of different processes to overlap. Due to the destructive aliasing from overlapping processes, performance deteriorates in all cases. We simulate this in the “CS” case by filling the predictor tables with spurious values (inverting all the bits in some cases and inserting random values in others) at the point when a context switch is scheduled to occur. We chose to compare predictors with a small size of 1K entries (except for the gshare and 2bit counter predictors which have more entries so that the hardware budget is same for all predictors) because it is typical for embedded processors to have small predictor sizes. Results are shown for the intervals of 100K, which gives a lower bound of the performance for the predictors, in the presence of context switches.

One point to note from the figure is that the prediction accuracy of certain predictors like skew and hybrid is more than that of the simple 2bit counter (bimodal) which is used in XScale, even for small hardware budgets. The reason for this is that these predictors handle aliasing and utilize global history better than the simple bimodal predictor to give a marked improvement in performance. The main observation from Figure 6 however is that the loss in prediction accuracy is significant for all of the predictors, due to the presence of context switches.

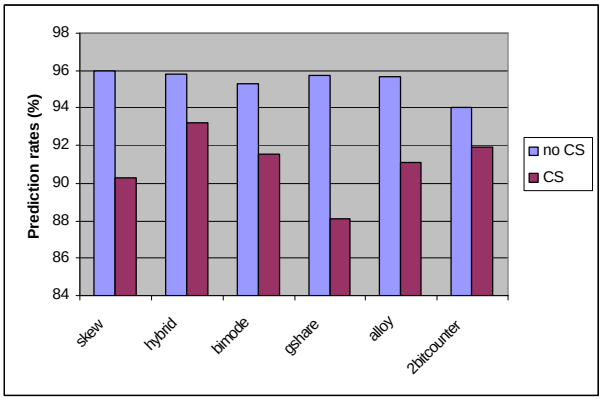


Figure 6. Predictor accuracy in the presence of context switches (6K bit budget, 100K CS interval)

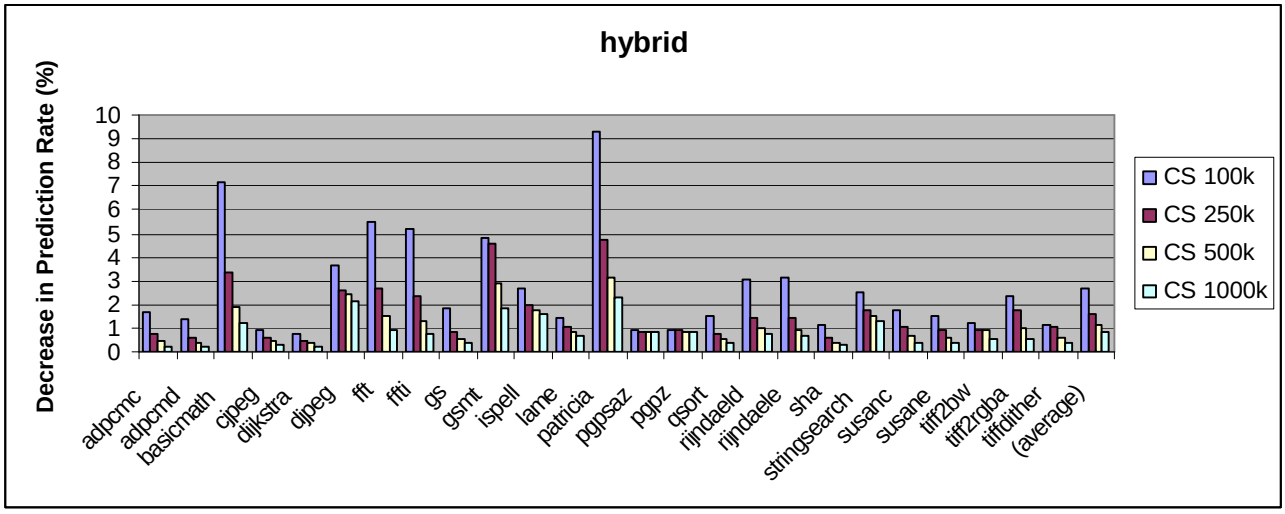
Figure 7 shows the performance degradation for the dynamic branch prediction schemes discussed in the previous section with changing context switch intervals. We chose a direction predictor table budget of 6K bits corresponding to 1K entries for all predictors except gshare and the 2bit counter as explained above, and context switch intervals of 100K, 250K, 500K and 1000K instructions, keeping in mind the numbers obtained for the context switch interval range from our experiments.

It is apparent from the figure that with an increase in context switch interval, performance for all the predictors improves. However the loss in prediction accuracy is still significant for many of the benchmarks even when the context switch interval is increased. Also low context switch intervals are quite frequently encountered in systems, and for these intervals the degradation in performance is large.

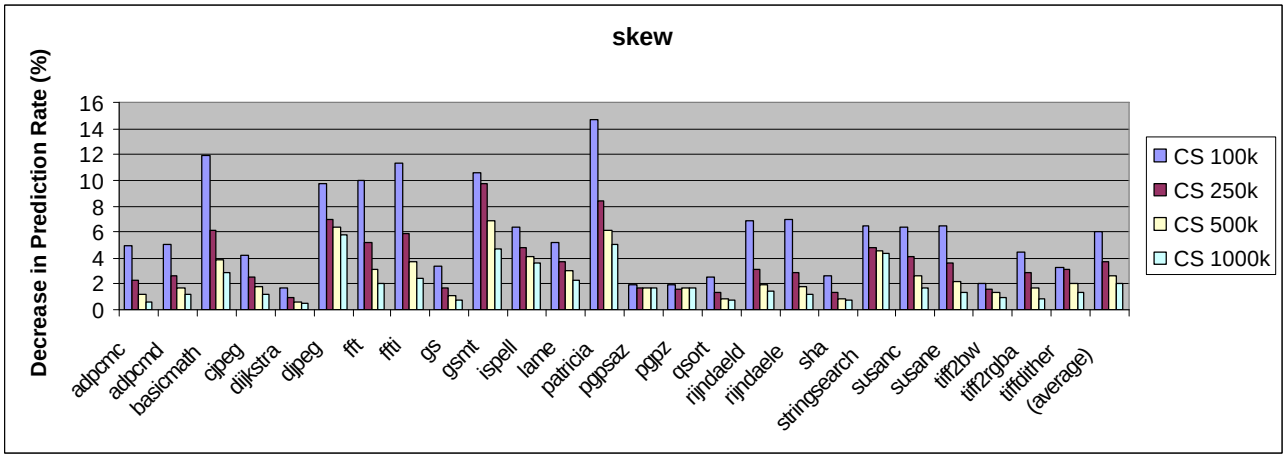
All of the predictors except the bimodal predictor rely heavily on the global history register to distribute prediction entries for branches and it takes time to train the predictors after a context switch, before correct predictions can be obtained. From Figure 7 it can be seen that for a context switch interval of 100K instructions, the misprediction rate for certain benchmarks like patricia, gsmt, fft and basicmath can be very high regardless of the dynamic predictor used, ranging from 5 to 17%! This shows how important it is to address the effect of context switches on prediction accuracy. The alloy predictor concatenates the address bits with the local and global history bits to index into the predictor table. Since a lot more address bits are used (7) compared to local (2) and global (2) history bits, the effect of erroneous entries in the history registers is not as much of a problem as it is with the skew and gshare predictors where the entire history register contents are xor'ed with the address. Therefore the decrease in prediction accuracy is slightly less for the alloy predictor. The prediction accuracy for the bimode predictor decreases comparatively less when compared to the alloy scheme because it uses a concatenation of address and global history bits to index into the predictor tables. Since there is no local history pattern for the bimode predictor, it is not affected by incorrect values in the local history predictor at the time of a context switch and the prediction accuracy does not decrease as much as for the alloy predictor. Studies [8, 9] confirm that the performance of single-scheme predictors, like the ones discussed above, deteriorates when branch history information is destroyed periodically. Hybrid predictors outperform these schemes because they have a bimodal component which takes very little time to warm up after a context switch. The longer warm-up time for the other component – the gshare predictor, does not affect performance because on a misprediction by gshare, the meta predictor table gets biased towards the fast training bimodal component, and this gives better prediction. However, as Figure 7(a) indicates, hybrid predictors still incur a substantial decrease ranging from 1 to 9% in their prediction rate. Finally, the 2 bit counter (bimodal) predictor is least effected by context switches because of reasons discussed earlier – since it is basically a table of 2 bit counters, it takes very little time to train it after a context switch, and consequently its prediction accuracy is not affected too much due to context switches.

An interesting observation from Figure 7 is that for a few benchmarks like pgpz and dijkstra, performance does not decrease very much. This is because they contain a small number of static branches that are executed over and over again. As a result, predictors can be quickly trained to give accurate predictions after a context switch, and the loss in performance is negligible. We are more interested in benchmarks that have large static and dynamic signatures [5].

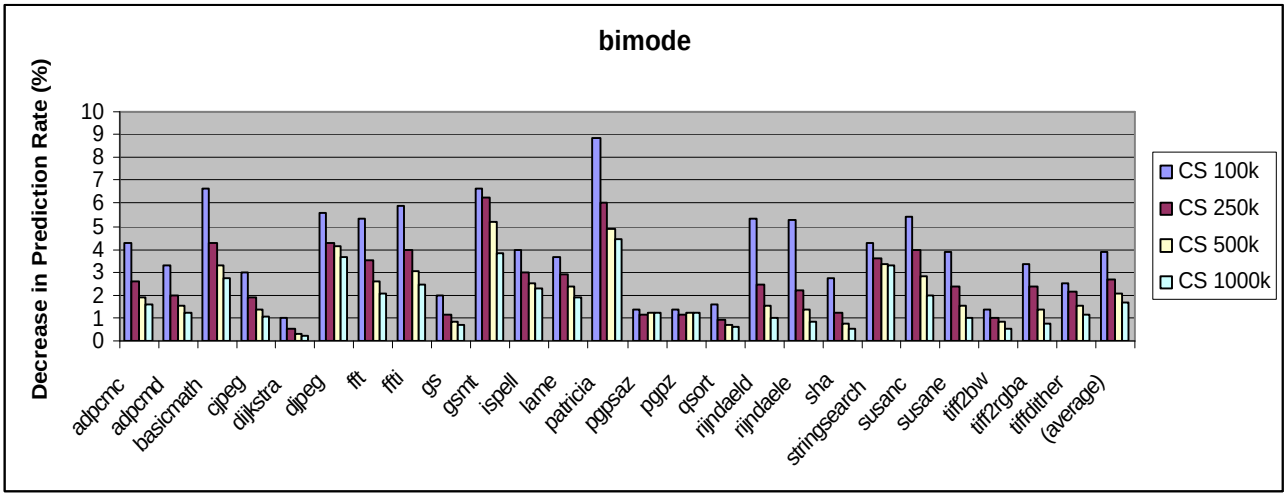
Figure 8 shows the decrease in average prediction accuracy for the cases when context switches are absent and when they are present, for different predictor sizes – 3k, 6k, 12k and 24k bits. Although increasing predictor size improves performance because larger predictor tables result in lesser destructive aliasing, the effect of context switches become more prominent with increasing predictor size and causes the prediction rate to decrease with increasing predictor size.



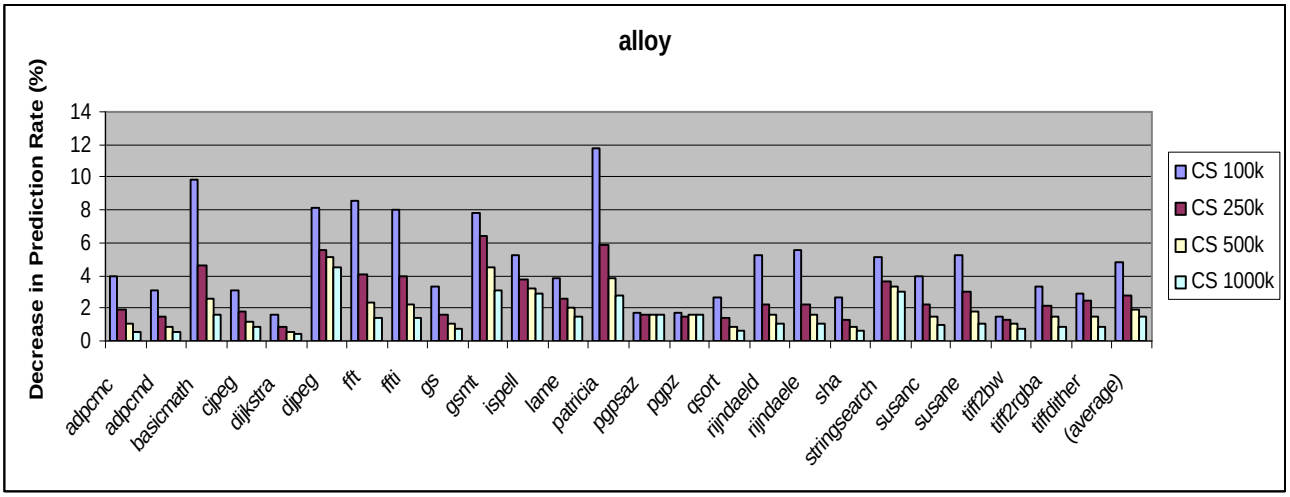
(a) Decrease in Prediction Rate for the Hybrid predictor for varying CS intervals



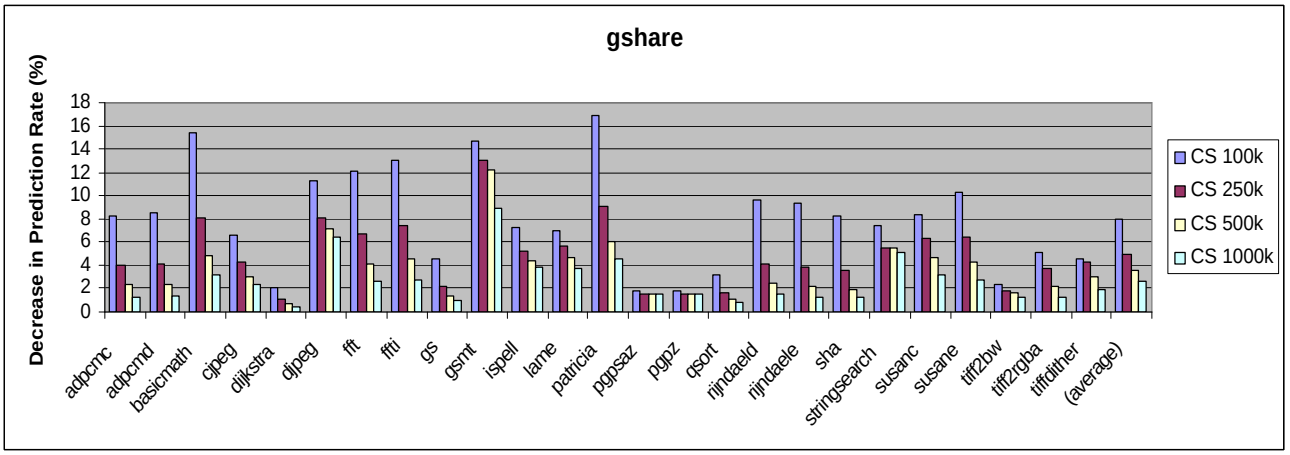
(b) Decrease in Prediction Rate for the Skew predictor for varying CS intervals



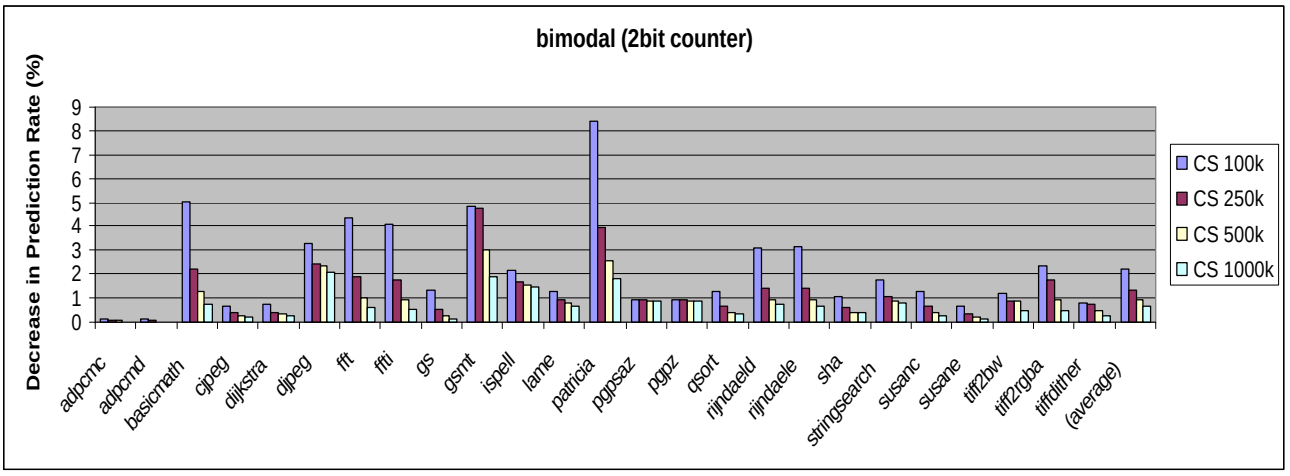
(c) Decrease in Prediction Rate for the Bimode predictor for varying CS intervals



(d) Decrease in Prediction Rate for the Alloy predictor for varying CS intervals



(e) Decrease in Prediction Rate for the Gshare predictor for varying CS intervals



(f) Decrease in Prediction Rate for the Bimodal (2bit counter) predictor for varying CS intervals

Figure 7. Decrease in Prediction Rate for different dynamic predictors with varying Context Switch Intervals

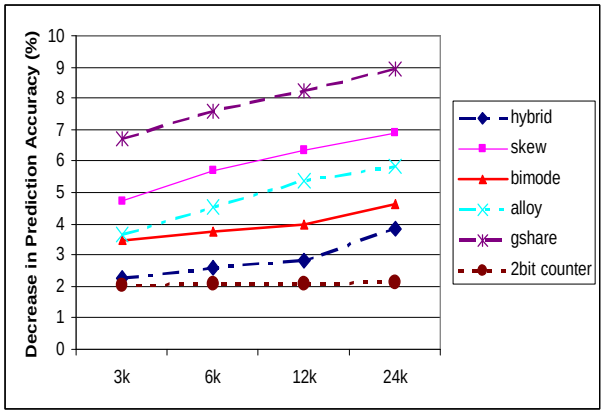


Figure 8. Percentage decrease in average prediction accuracy due to context switches for dynamic predictors with varying hardware budgets (100K CS interval)

Future generations of embedded processors are expected to have deeper pipelines to exploit parallelism and consequently larger predictor sizes to improve prediction rates, since misprediction penalty with deeper pipelines will be more. As predictor sizes increase, the degradation in performance due to context switches will become more pronounced as can be clearly seen in Figure 8. In the next section we propose schemes to overcome this performance loss due to context switches.

5. New Schemes to Improve Accuracy

On a context switch, the predictor structure contains information for the process that just finished execution. This information does not accurately represent predictions for other processes, as was shown in Figure 6. One existing scheme is to flush the predictor bits to zero every time a context switch occurs [20]. Another scheme [3] sets all the predictor table entries to weakly taken. This section describes new schemes we use to improve on the above.

An "alternating" scheme sets the counter bits alternatively to weakly taken and not-taken (Figure 9) for successive counters in the table. This scheme has been used for initializing a predictor but has not been applied in the presence of context switches. The weakly taken and alternating schemes are effective because studies [16-18] show that more than half of all branches. tend to be taken, and branches in general tend to be biased towards either being taken or not-taken. By setting the counters to a weak bias (01 or 10) instead of a strong bias (00 or 11), we allow a branch to quickly revert to the opposite prediction if it is biased in that direction. For instance, if a branch is biased towards not-taken and we set the counter to weakly taken after a context switch, it will take just one misprediction before we can get a correct prediction for that branch. However if we had set the counter to strongly taken after the context switch, it would take two mispredictions before we could get a correct prediction. All of these schemes require very little overhead and can be considered to be static in nature because they bias the predictor tables based on decisions which do not take into account the dynamic state of the predictor when executing.

If there were a way to save the entire prediction information from the predictor tables for a process, and then restore the predictions into the predictor structure when the process resumes execution, there would be no loss in performance. However, saving and restoring entire predictor structures as was done in [3] can be prohibitive both in terms of time and memory space. Therefore we investigate several schemes for saving "representative" portions of the prediction information. For instance, one simple scheme we propose saves 1 bit per table for each process when a context switch occurs. This bit is the predominant bias of the predictor tables – either taken or not taken. When the process resumes execution, the bit is used to bias the counters in the predictor tables based on its value. We call this the *majority bias* scheme.

Another scheme "compresses" the predictor state and saves N bits per process. The value of N is selected so as to achieve a desired accuracy improvement without large overhead. A compression algorithm partitions a predictor table into blocks of *k* entries and stores the state information for each block. The state can be the dominant bias bit for a block. Alternatively, 2 bits of state can be saved per block (the scheme evaluated in this paper). On a context switch, the number of entries in the block set to strongly not-taken, weakly not-taken, weakly taken and strongly taken is used to save the state of a block as following:

?? 00: if there are more strongly not-taken entries than strongly taken entries in the block. If these are equal then the weakly not-taken and weakly taken entries are compared and a 00 is saved if there are more weakly not-taken entries.

?? 01: if there are more strongly taken entries than strongly not-taken entries. If these are equal then the weakly taken and weakly not-taken entries are compared and a 01 is saved if there are more weakly taken entries.

?? 10: otherwise if the overall number of taken and not-taken entries is the same, we save a 10.

When the process is resumed and before it commences execution, the two saved bits for each block of the predictor table are used to restore the state as follows: If 00 was stored, we bias all the counters in the block to weakly not-taken. If the saved value was 01, we bias all the counters to weakly taken. For the case of a 10, we bias successive counters in the tables alternately to weakly taken and not-taken. To implement this scheme,

hardware counters are used together with some combinational logic to route and store the data.

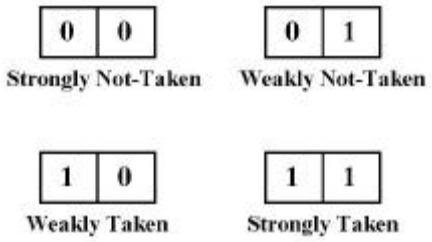


Figure 9: States of a 2-bit counter

Another way to reduce the amount of predictor information saved would be to save just the most significant bit of the two bit counters in the tables as proposed in [3]. This *snapshot* of the most significant bits of the counters in the tables reduces the information stored by half. Instead of saving snapshots for all the tables, a *partial snapshot* can be taken for a subset of the tables in the predictor. This further reduces the information stored, at the cost of reduced prediction accuracy.

To save and restore the predictor information, we propose the use of an L2 cache, bypassing the smaller L1 caches, or alternately we can also use a small dedicated buffer for the purpose. The results in the next section show that schemes which require saving as little as 96 bits for a 1K entry predictor can achieve a significant improvement in prediction rate (up to 6%) for the hybrid predictor which has one the best prediction rate in the absence of context switches compared to the other dynamic predictors selected for study. The time penalty for doing this is a few extra cycles for saving and restoring the information, including the combinational logic delay for compression. This is a small price to pay when compared with the large improvement in performance gained from more accurate branch prediction. It is also very small compared to the overall context switch overhead and can be done entirely in hardware. Recall that the minimum penalty on a branch misprediction for the Xscale processor is 4 cycles. Thus if a scheme exhibits even a marginal improvement in prediction rate, it can justify the overhead of saving and restoring the predictor information.

6. Performance Evaluation

Our goal was to reduce the prediction accuracy loss due to context switches for dynamic branch predictors. We choose the hybrid predictor [12] because it provides one of the best performances for the 1K entry predictors considered and because it is widely used, but the proposed schemes work for other dynamic predictors too.

All simulations were performed on a modified version of the *sim-outorder* simulator from SimpleScalar [19] version 3.0. The processor modeled uses a configuration similar to the Xscale [20] processor: 32KB data and instruction L1 caches with 32 byte lines and 1 cycle latency, no L2 cache and 50 cycle main memory access latency. The machine is in-order and a 32 entry load/store queue, with an issue width of 2. It has one integer unit, one floating point unit and one multiply/divide unit. The branch predictor has 128 entries, and the instruction and data TLBs are 32 entry and fully associative.

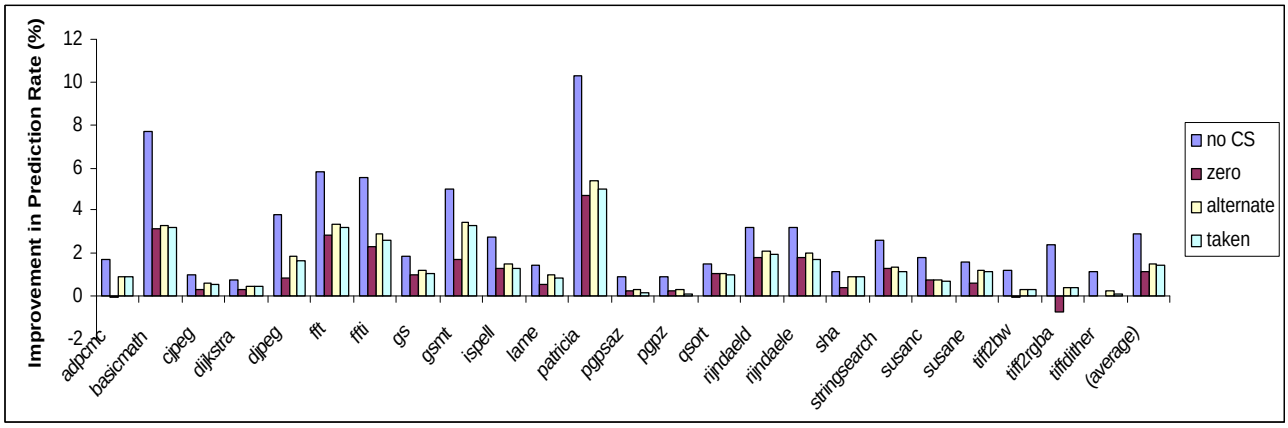
We selected 24 benchmarks from the MiBench [21] suite which is designed to be representative of several embedded systems domains. These include benchmarks from various embedded systems domains: basicmath, qsort, susan (automotive and control), djpeg, lame, tiff2bw, tiff2rgba, tiffdither (consumer), ghostscript, ispell, stringsearch (office automation), sha, dijkstra, patricia (network), pgp, rijndael (security), fft, adpcm and gsm (telecommunications).

A context switch interval of 100K instructions was chosen to represent a lower bound on performance when context switches are present. A small 1K entry hybrid predictor was used. All simulations were run till termination.

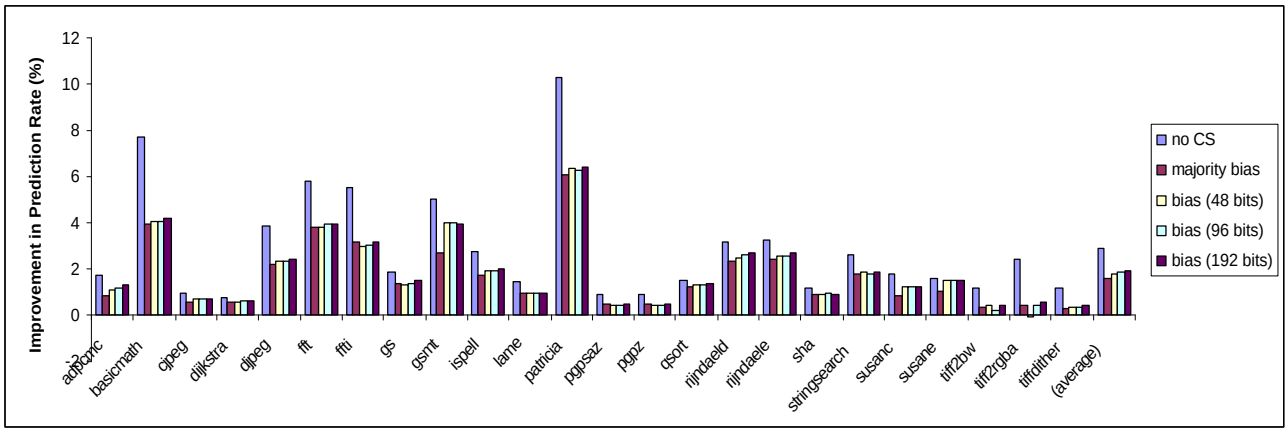
6.1. Results

Figure 6 showed the hybrid predictor to have a significant loss of accuracy with context switches for several benchmarks. This section shows the effect of various schemes that change the predictor state on a context switch.

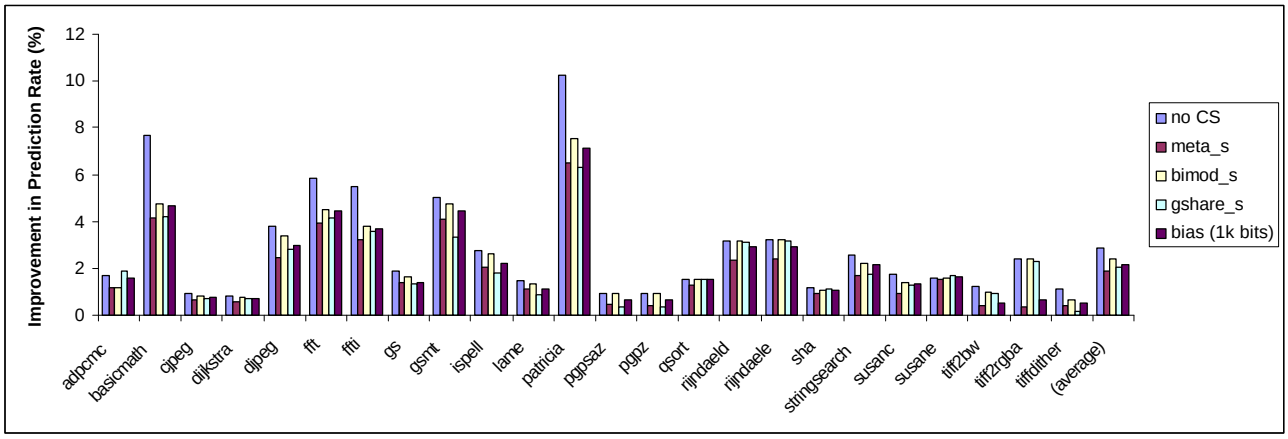
First, the schemes that do not save or restore predictor information are evaluated. They are referred to as "negligible overhead" schemes. Figure 10(a) shows the improvement in predictor accuracy for the hybrid predictor when these schemes are used. The “no CS” case is an upper bound on the improvement and corresponds to saving all of the (6K) bits of the hybrid predictor. As expected, biasing the predictor tables to weakly taken [3] is better than flushing them to zero because branches in general are more inclined towards being taken. One of the schemes proposed in this paper is also shown. It sets the table counters alternately to weakly taken (10) and weakly not-taken (01) and provides a slightly better performance than the weakly taken scheme. The weakly taken scheme mispredicts the case when a branch is highly biased towards not being taken – for instance the branch condition in a *for* loop.



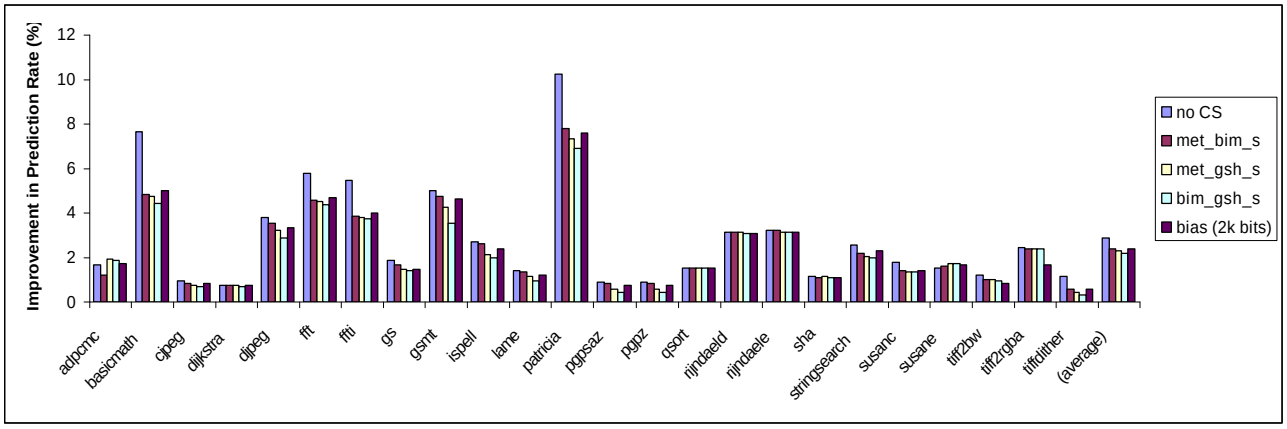
(a) "negligible overhead" schemes



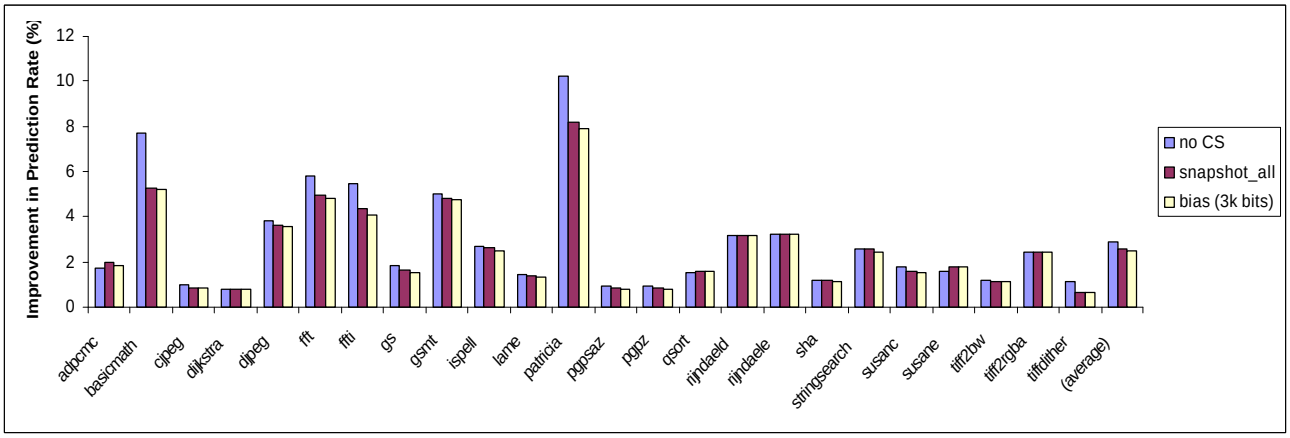
(b) "low overhead" schemes



(c) "medium overhead" schemes



(d) "high overhead" schemes



(e) "very high overhead" schemes

Figure 10. Improvement in prediction rates for the proposed schemes, in the presence of context switches (1K entry hybrid predictor, 100K CS interval)

Next we demonstrate the effect of saving[1] predictor information to improve prediction accuracy. Depending upon the amount of information saved, the schemes are divided into four categories – low, medium, high and very high overhead schemes. Figure 10(b) shows the performance improvement for the "low overhead" schemes. The majority bias scheme saves one bit per table. The other three schemes save 2 bits for a block of k entries as explained earlier. For a 1K entry table the value of k is 64 when one word per table (32 bits/table or 96 bits overall) is saved, 32 when 2 words are saved (192 bits overall) etc. The prediction accuracy improves when more bits are saved. These schemes perform better than all of the "negligible overhead" schemes because they use dynamic information.

Figure 10(c) shows the improvement of "medium overhead" schemes. A *partial snapshot* scheme is used (compared to the snapshot-all scheme from [3]) for one table while the weakly taken scheme is used for the other two tables. These schemes require saving and restoring 1K bits of information for each process. The scheme that stores the snapshot of the bias bits of the bimodal table is better than the other schemes. This indicates that the bimodal component is more important and predicts branch direction for more branches than the gshare component. The bias scheme with a budget of 1K bits is shown as well. This scheme saves information for all three tables and thus uses compression and looses accuracy. It does not perform as well as the bimodal snapshot scheme. It is also interesting to note that the performance for the case when we save the bias bits of the meta table (1K bits) is the same as the case when we save just 1 word/table (96 bits overall)! This shows how important it is to carefully select the bits to save for maximum performance gain.

[1] In this and all subsequent schemes that involve saving predictor information for the gshare component of the hybrid predictor, we will assume that the 10 bits of the global history register are also saved.

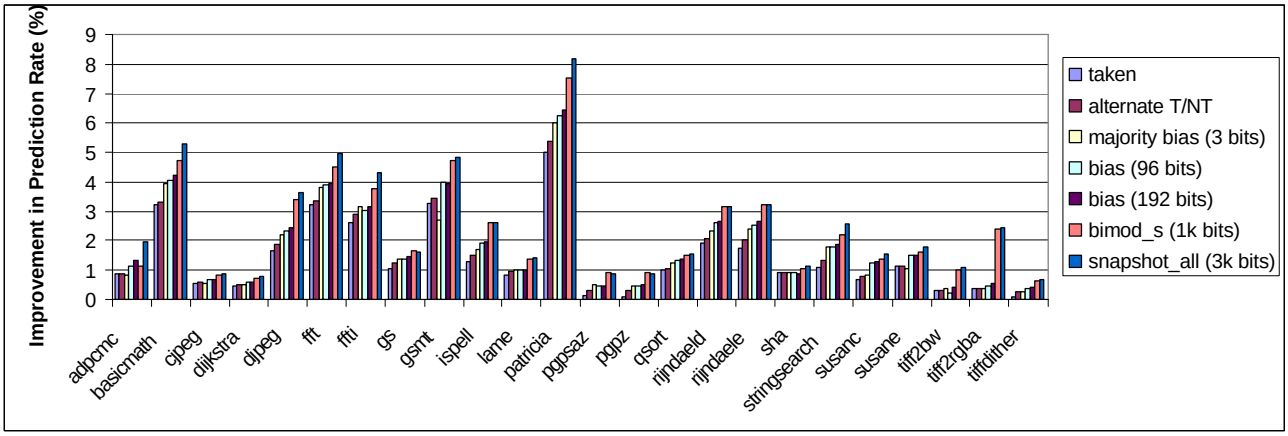


Figure 11. Improvement in prediction rate for selected schemes and benchmarks

Next the change in performance for the snapshot scheme with bias bits of two tables is analyzed. The bias scheme with a comparable budget is also presented in Figure 10(d). 2K bits are now saved and restored. This is a large amount of information and these schemes are classified as "high overhead" schemes. The maximum performance is achieved for the case where we save a snapshot of the bimodal and meta tables. This is in line with the previous results where the bimodal component was found to be the more important component of the two. The bias scheme performs well but slightly worse than the bimodal meta snapshot scheme which is the best scheme in the category.

Finally, Figure 10(e) shows the performance of the "very high overhead" schemes which save and restore one half of the predictor structure (3K bits). Not unexpectedly, this is enough information to be very accurate as the snapshot scheme in [3] has shown. The bias scheme with a similar budget does not perform as well.

An interesting observation from Figure 10 is that the bias scheme works well for small budgets and does not scale well below a certain block size *k*. Overall, the "low overhead" bias schemes saving less than 200 bits, give an average improvement of 1.9%. If more information can be saved, the partial snapshot scheme is preferable over the bias scheme. Saving a snapshot of the bimodal table and setting the counters to weakly taken for the other two tables results in an improvement of around 2.4%. This is within 0.5% of the case when no context switches occur.

Figure 11 compares the weakly taken and snapshot-all schemes from [3] with five schemes proposed in this paper: the alternate weakly taken/not-taken scheme, the majority bias scheme with a budget of 1 bit/table, the bias scheme with a budget of 96 bits (1 word/table) and 192 bits (2 words/table) and the partial snapshot scheme for the bimodal table. These schemes are chosen as the best representatives in the "practical" overhead categories – the bias scheme saving 96 bits is included since it is very close in performance to the bias scheme which saves 192 bits. It can be seen that on an average, the alternate taken/not-taken scheme outperforms the weakly taken scheme. For the benchmarks shown, the low overhead bias schemes are better than the two negligible overhead schemes by 0.5 – 1% in several cases. The bimode snapshot scheme is better than this bias scheme by 0.5 – 1% on average, and is within 0.3% of the improvement from snapshot-all on an average. This shows how useful the proposed schemes are in improving performance for certain applications – they save and restore a small amount of information in a few cycles to achieve significant improvement in performance. The variation in the gain in performance on using these schemes implies that the designer must carefully consider the performance/overhead ratio before deciding to implement any of the proposed schemes.

7. Summary and Conclusion

This paper evaluated the loss of prediction accuracy in the presence of context switches for several branch predictors. The loss of accuracy was shown to be significant for all the considered predictors. Several new mechanisms to restore the lost accuracy for the hybrid predictor were presented. The hybrid predictor was selected because it is widely used and the new schemes evaluated assuming a 100K instructions context switch interval. The latter is a lower bound on observed context switch interval size.

Most of the proposed schemes involve saving and restoring varying amounts of predictor information on a context switch. This entails an overhead which needs to be considered in selecting a practical scheme. It was shown that for the *bias* scheme, saving 96 (table)+10 (global register) bits can improve the prediction rate from 1 - 6%. The *majority bias* scheme performs slightly

worse than this scheme on the average, but saves just 3 (table)+10(global register) bits to achieve improvement. Both these schemes perform significantly better then the previously-proposed low-overhead schemes - flush to *zero* and weakly *taken*. Overall, we do not consider the snapshot schemes practical to implement or their additional performance improvement worthwhile.

This paper concentrates on prediction accuracy and does not evaluate the impact on overall CPU performance. Thus we do not describe the design of the new schemes or their latency in much detail. We believe that it can be done efficiently for the low overhead schemes where the information to be stored is continuously generated from the predictor data by dedicated logic. Therefore only saving/restoring the compressed information takes time. This may be doable purely in hardware without executing additional instructions.

These proposed mechanisms are not limited to the hybrid scheme and can be used effectively with other dynamic prediction schemes. Similar performance improvements can be expected for the skew, bimode, and gshare predictors. In fact, any dynamic predictor using a bimodal table structure can benefit from our schemes. Schemes that involve large pattern history tables such as the alloy predictor are harder to deal with when it comes to reducing the information saved and restored on a context switch. This remains subject of future research.

8. References

[1] Michele Co., K. Skadron "The Effects of Context Switching on Branch Predictor Performance". In *Proceedings of the 2001 IEEE International Symposium for Performance Analysis of Systems and Software*, November, 2001, Tuscon, AZ

[2] Marius Evers, Po-Yung Chang, Yale N. Patt "Using Hybrid Branch Predictors to Improve Branch Prediction Accuracy in the Presence of Context Switches". In *Proceedings of the 23rd Intl. Sym. on Computer Architecture*, pp. 3-11, 1996.

[3] Ashutosh S. Dhodapkar and James E. Smith "Saving and Restoring Implementation Contexts with co-Designed Virtual Machines". In *Workshop on Complexity-Effective Design*, June 30 2001, Goteborg, Sweden.

[4] P.-Y. Chang, M. Evers, and Y. Patt. "Improving Branch Prediction Accuracy by Reducing Pattern History Table Interference". *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques*, Oct. 1996.

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

[6] Tse-Yu Yeh, Yale N. Patt, "Alternative Implementations of Two-Level Adaptive Branch Prediction" In *Nineteenth International Symposium on Computer Architecture*, 1992

[7] Nicolas Gloy, Cliff Young, J. Bradley Chen, Michael D. Smith, "An Analysis of Dynamic Branch Prediction Schemes on System Workloads", In *Proc. 23rd Annual Intl. Symp. on Computer Architecture*, 1996

[8] R. Nair. "Dynamic Path-Based Branch Correlation". In *28th International Symposium on Microarchitecture*, pages 15--23, November 1995.

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

[10] T.Juan, S.Sanjeevan and J.J.Navarro, "Dynamic History-Length Fitting: A third level of adaptivity for branch prediction", In *Proceedings of the 25th Annual Intl. Symposium on Computer Architecture*, pp 155-166, June 1998.

[11] K. Skadron et al., "A taxonomy of branch mispredictions, and alloyed prediction as a robust solution to wrong-history mispredictions," In *PACT 2000*, pp. 199-206.

[12] S. McFarling. "Combining branch predictors". In *DEC WRL TN-36*, June 1993.

[13] C.-C. Lee, I.-C. Chen, and T. Mudge. "The bi-mode branch predictor". In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, Dec 1997.

[14] S. T. Pan, K. So, and J. T. Rahmeh. "Improving the accuracy of dynamic branch prediction using branch correlation" In *Proceedings of ASPLOS V*, pages 76–84, Boston, MA, October 1992.

[15] P. Michaud, A. Seznec, and R. Uhlig, "Trading conflict and capacity aliasing in conditional branch predictors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 292--303, 1997

[16] P. Chang, E. Hao, T. Yeh, and Y. Patt. "Branch classification: a new mechanism for improving branch predictor performance". In *MICRO-27*, November 1994.

[17] J. Smith. "A study of branch prediction strategies". In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, May 1981.

[18] Johny Lee and Alan Smith. "Branch prediction strategies and branch target buffer design" In *Computer, 17(1):6-22*, 1984. 18

[19] D. Burger and T. M. Austin. "The SimpleScalar tool set, version 2.0." Technical Report 1342, *Computer Sciences Department*, University of Wisconsin-Madison, June 1997

[20] Intel, *Intel Xscale Microarchitecture*, 2001

[21] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin,

T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, pages 83–94, 2001.