31st Annual IEEE/ACM Symposium on Microarchitecture. December, 1998. pp. 69-77

The YAGS Branch Prediction Scheme

A. N. Eden and T. Mudge, {ane, tnm}@eecs.umich.edu
Dept. EECS, University of Michigan, Ann Arbor

Abstract

The importance of an accurate branch prediction mechanism has been well documented. Since the introduction of gshare [1] and the observation that aliasing in the PHT is a major factor in reducing prediction accuracy [2,3,4,5], several schemes have been proposed to reduce aliasing in the PHT [6, 7, 8, 9]. All these schemes are aimed at maximizing the prediction accuracy with the fewest resources. In this paper we introduce Yet Another Global Scheme (YAGS) — *a new scheme to reduce the aliasing in the PHT — that combines the strong points of several previous schemes. YAGS introduces tags into the PHT that allows it to be reduced without sacrificing key branch outcome information. The size reduction more than offsets the cost of the tags. Our experimental results show that YAGS gives better prediction accuracy for the SPEC95 benchmark suite than several leading prediction schemes, for the same cost. It also performs better than the other schemes in the presence of a context switch. Finally, YAGS displays good results for the go benchmark, which is of special interest since it has a large number of static branches and reflects situations where aliasing in the PHT can be a problem.*

1. Introduction

To realize the performance potential of today's widely-issued, deeply pipelined superscalar processors, a good branch prediction mechanism is essential. The introduction of two level adaptive schemes was an important step in this direction [10]. They are able to achieve predicted levels of 90% or more. Of the two level schemes, global history schemes appear to work best for integer code [11]. This, in part, is due to the large number of if-else instructions in integer programs. Sequences of if-else are often highly correlated.

The main problem which reduces the prediction rate in the global schemes is aliasing between two indices (an index is typically formed from history and address bits) that map to the same entry in the Pattern History Table (PHT). Since the information stored in the PHT entries is either "taken" or "not taken," two aliased whose corresponding information is the same will not result in mispredictions. We define this situation as neutral aliasing. On the other hand, two aliased indices with different information might interfere with each other and result in a misprediction. We define this situation as destructive aliasing. This paper is organized as follows: the next section looks at previous schemes to reduce aliasing and highlights their strong and weak points. In the third section we introduce Yet Another Global Scheme (YAGS), which combines the strengths of the previous schemes to eliminate aliasing. The fourth section presents the results of our performance studies. The fifth section offers concluding remarks and proposes future directions for this research.

2. Previous Work

Gshare. The first scheme to address the aliasing problem in two level adaptive branch predictors was gshare [1] (figure 1). The observation that the usage of the PHT entries is not uniform when indexed by concatenations of the global history and the branch address, led to idea of using the "exclusive or" function instead of concatenation to more evenly use the entries in the PHT. Detailed studies have shown it yields little, if any, advantage [4].

The Agree Predictor. The agree predictor (figure 2) assigns a biasing bit to each branch in the Branch Target Buffer (BTB) according to the branch direction just before it is written into the BTB [7]. The PHT information is then changed from "taken" or "not taken" to "agree" or "disagree" with the prediction of the biasing bit. The idea behind the agree predictor is that most branches are highly biased to be either taken or not taken and the hope is that the first time a branch is introduced into the BTB it will exhibit its biased behavior. If this is the case, most entries in the PHT will be "agreeing," so if aliasing does occur it will more likely be neutral aliasing, which will not result in a misprediction.

It is one of the first two level scheme to take advantage branches' biased behavior to reduce destructive aliasing by replacing it with neutral aliasing. It considerably reduces destructive aliasing. However, there is no guarantee that the first time a branch is introduced to the BTB its behavior

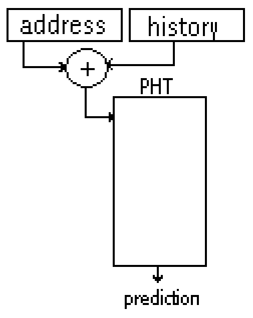

Figure 1. Gshare

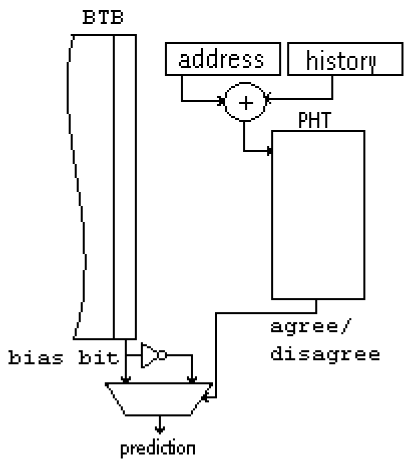

Figure 2. Agree

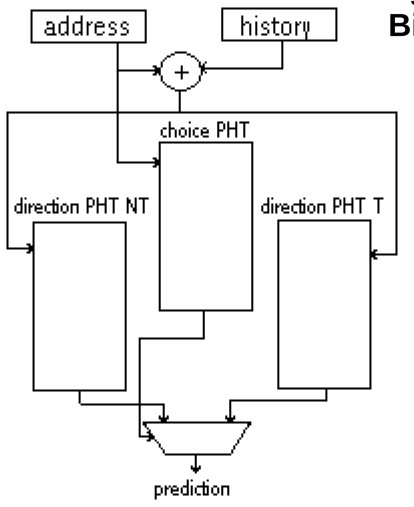

Figure 3. Bi-Mode

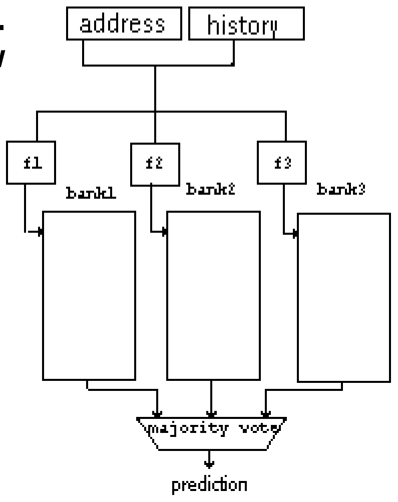

Figure 4. Skew

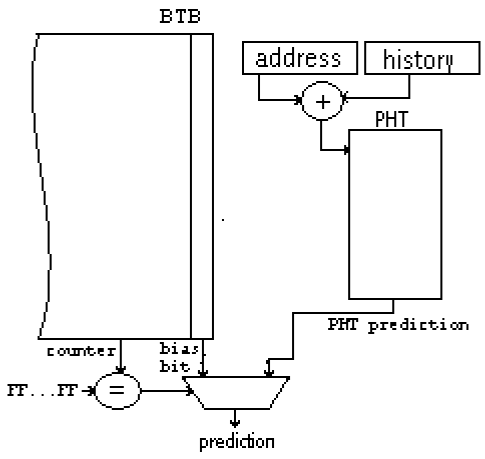

Figure 5. Filter

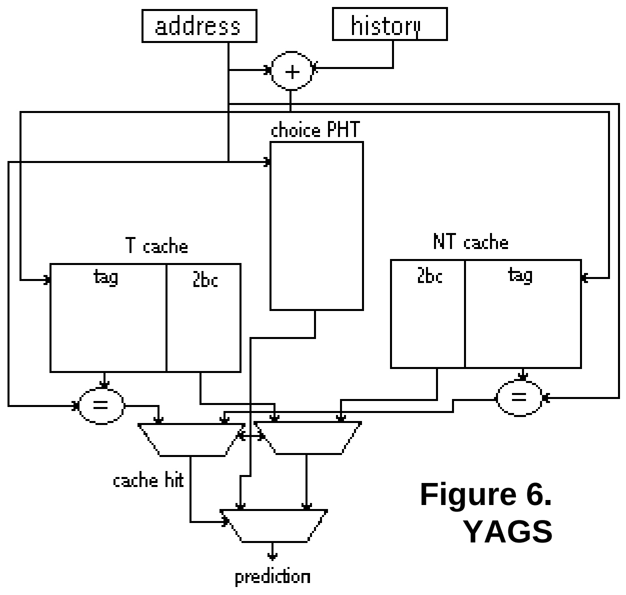

Figure 6. YAGS

will correspond to its bias. When such cases occur, the biasing bit will stay the same until the branch is replaced in the BTB by a different branch. Meanwhile, it will pollute the PHT with “disagree” information. There is still aliasing between instances of a branch which do not comply with the bias and instances which do comply with the bias. Furthermore, when a branch is not in the BTB, no prediction is available.

The Bi-Mode Predictor. The bi-mode predictor (figure 3) tries to replace destructive aliasing with neutral aliasing in a different manner [8]. It splits the PHT table into even parts. One of the parts is the choice PHT, which is just a bimodal predictor (an array of two bit saturating counters) with a slight change in the updating procedure. The other two parts are direction PHTs; one is a “taken” direction PHT and the other is a “not taken” direction PHT. The direction PHTs are indexed by the branch address xored with the global history. When a branch is present, its address points to the choice PHT entry which in turn chooses between the “taken” direction PHT and the “not taken” direction PHT. The prediction of the direction PHT chosen by the choice PHT serves as the prediction. Only the direction PHT chosen by the choice PHT is updated. The choice PHT is normally updated too, but not if it gives a prediction contradicting the branch outcome and the direction PHT chosen gives the correct prediction.

As a result of this scheme, branches which are biased to be taken will have their predictions in the “taken” direction PHT, and branches which are biased not to be taken will have their predictions in the “not taken” direction PHT. So at any given time most of the information stored in the “taken” direction PHT entries is “taken” and any aliasing is more likely not to be destructive. The same phenomenon happens in the “not taken” direction PHT. The choice PHT serves to dynamically choose the branches’ biases.

In contrast to the agree predictor, if the bias is incorrectly chosen the first time the branch is introduced to the BTB, it is not bound to stay that way while the branch is in the BTB and as a result pollute the direction PHTs.

However, the choice PHT takes a third of all PHT resources just to dynamically determine the bias. It also does not solve the aliasing problem between instances of a branch which do not agree with the bias and instances which do.

The Skewed Branch Predictor. The skewed branch predictor (figure 4) is based on the observation that most aliasing occurs not because the size of the PHT is too small, but because of a lack of associativity in the PHT (the major contributor to aliasing is conflict aliasing and not capacity aliasing). The best way to deal with conflict aliasing is to make the PHT set-associative, but this requires tags and is not cost-effective. Instead, the skewed predictor emulates associativity using a special skewing function [6].

The skewed branch predictor splits the PHT into three even banks and hashes each index to a 2-bit saturating counter in each bank using a unique hashing function per bank (f1, f2 and f3). The prediction is made according to a majority vote among the three banks. If the prediction is wrong all three banks are updated. If the prediction is correct, only the banks that made a correct prediction will be updated (partial updating).

The skewing function should have inter-bank dispersion. This is needed to make sure that if a branch is aliased in one bank it will not be aliased in the other two banks, so the majority vote will produce an unaliased prediction.

The reasoning behind partial updating is that if a bank gives a misprediction while the other two give correct predictions, the bank with the misprediction probably holds information which belongs to a different branch. In order to maintain the accuracy of the other branch, this bank is not updated.

The skewed branch predictor tries to eliminate all aliasing instances and therefore all destructive aliasing. Unlike the other methods, it tries to eliminate destructive aliasing between branch instances which obey the bias and those which do not. However, to achieve this, the skewed predictor stores each branch outcome in two or three banks. This redundancy of 1/3 to 2/3 of the PHT size creates capacity aliasing but eliminates much more conflict aliasing, resulting in a lower misprediction rate. However, it is slow to warm-up on context switches.

The Filter Mechanism. Reducing the amount of redundant information stored in the PHT is the main point of this scheme [9]. The idea is that highly biased branches can be predicted with high accuracy with just one bit. The filtering of such branches out of the PHT is done by a bias bit and a saturating counter (figure 5) for each BTB entry. When a branch is introduced to the BTB the bias bit is set to the direction of the branch when it is resolved and the counter is initialized. When every branch instance is resolved, if the direction of the branch is the same as the bias bit the counter is incremented. If not, the counter is zeroed and the bias bit is toggled. A branch is predicted using the PHT if the counter is not saturated. If the counter is saturated, it means that the branch is highly biased in the direction indicated by the bias bit, and therefore the bias bit is used as a prediction. In this case, when the counter is saturated, the PHT is not updated with the branch outcome

— the saturated counter filters this information from the PHT.

The size of the counter has to be tuned to the size of the PHT. If the PHT size is large, the amount of filtering needed is small, and therefore the size of the counters should be large.

When a branch is first introduced in the BTB, the counter is initialized. It was found that it is best to initialize the counter to its maximum value so the filtering mechanism will start working immediately. If the branch is not highly biased, the bias bit will flip fairly quickly and the counter will be zeroed. On the other hand, if the counter is initialized to zero and the branch is highly biased, it will take time for the filtering mechanism to start working and the PHT will be polluted in the meantime.

The filter mechanism tries to eliminate all aliasing instances, neutral and destructive, by considerably reducing the amount of information stored in the PHT. However, it mispredicts instances of highly biased branches which do not comply with the bias.

3. YAGS

The brief overview above, of earlier proposals to reduce aliasing in global schemes, suggests that splitting the PHT into two branch streams corresponding to biases of "taken" and "not taken," as is done in the agree and bi-mode predictors, is a good idea. However, as in the skewed branch predictor, we do not want to neglect aliasing between biased branches and their instances which do not comply with the bias. Finally, it will be beneficial if we can reduce the amount of unnecessary information in the PHT, as in the filter mechanism, but not at the expense of mispredicting some of the branch instances.

The motivation behind YAGS is the observation that for each branch we need to store its bias and the instances when it does not agree with it (figure 6). If we employ a bimodal predictor to store the bias, as the choice predictor does in the bi-mode scheme, than all we need to store in the direction PHTs are the instances when the branch does not comply with its bias. This reduces the amount of information stored in the direction PHTs, and therefore the direction PHTs can be smaller than the choice PHT. To identify those instances in the direction PHTs we add small tags (6-8 bits) to each entry, referring to them now as direction caches. These tags store the least significant bits of the branch address and they virtually eliminate aliasing between two consecutive branches.

When a branch occurs in the instruction stream, the choice PHT is accessed. If the choice PHT indicated "taken," the "not taken" cache is accessed to check if it is a special case where the prediction does not agree with the bias. If there is a miss in the "not taken" cache, the choice PHT is used as a prediction. If there is a hit in the "not taken" cache it supplies the prediction. A similar set of actions is taken if the choice PHT indicates "not taken," but this time the check is done in the "taken" cache. The choice PHT is addressed and updated as in the bi-mode choice PHT. The "not taken" cache is updated if a prediction from it was used. It is also updated if the choice PHT is indicating "taken" and the branch outcome was "not taken." The same happens with the "taken" cache.

We still need to take care of aliasing for instances of a branch which do not agree with the branch's bias. After making the introduction of tags cost-effective, the natural solution for the aliasing problem is to add associativity (in [6] it was showed that the vast majority of aliasing in the PHT is conflict aliasing).

When making the direction caches set-associative, there is some extra cost for keeping a correct replacement policy. For example, in a two-way set-associative cache, one bit for every two entries will suffice to keep track of which entry was replaced last. We use an LRU replacement policy with one exception: an entry in the "taken" cache which indicates "not taken" will be replaced first to avoid redundant information. If an entry in the "taken" cache indicates "not taken," this information is already in the choice PHT and therefore is redundant and can be replaced.

4. Performance Studies

4.1 Methodology

The experimental data presented in this paper were collected using SPEC95 benchmark traces. The benchmarks were compiled on the SunOS operating system using the gcc compiler. The traces were run to completion. In order to simulate a context switch for the context switch study only, a new trace file was created by interleaving all eight SPEC95 benchmarks every 60,000 instructions until one of the files runs out of instructions The number was chosen not to reflect a real context switching interval, which would be much less frequent, but to emphasize the effect of context switching on the various predictors. The size of the YAGS predictors includes the tags of the direction caches. In the case where YAGS is set-associative the LRU and history bits are also added.

4.2 Results

Figure 7 shows the misprediction rate for gshare, the skewed predictor, the bi-mode predictor and YAGS with direct mapped direction caches. As can be seen, YAGS performs better than the other schemes, particularly for small sizes. However, as the size of the PHT increases, YAGS's advantage over the other schemes decreases. This is to be expected, because, the aliasing problem in the PHT decreases with size and therefore the performance of all the schemes converges.

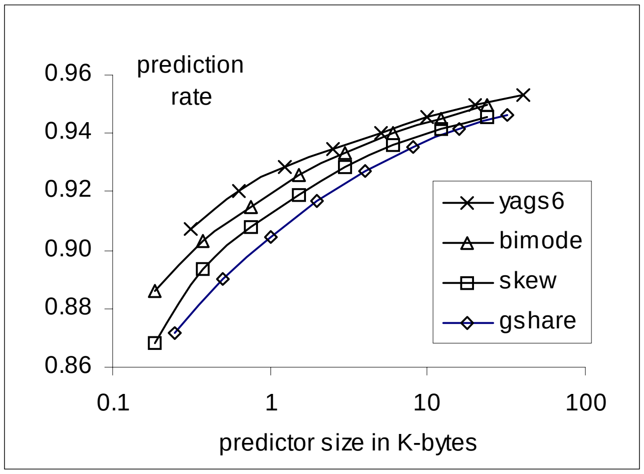

Figure 7. Prediction rates for four schemes including YAGS6 (6 bits in the tags).

One of the pitfalls of the SPEC95 benchmark suite is that most traces have a small static branch signature [8]. For example, the compress benchmark has only 482 static branches. These branches are executed over and over again throughout the course of the program. However, the small static branch signature implies each branch is more likely to have a unique entry in the PHT for each history instance, resulting in a very small amount of aliasing in the PHT. This yields optimistic figures for many branch predictions schemes.

The gcc and go benchmarks are thus of special interest because of their large static and dynamic branch signatures. As can be seen in figures 8 and 9, YAGS also outperforms the other schemes for the go and gcc benchmarks. The go benchmark is particularly interesting because it suffers the most from destructive aliasing. The gshare scheme for small predictors achieves a 69% correct prediction rate for go. For about the same amount of resources (0.5KB) YAGS achieves a 77% correct prediction rate. The bi-mode, which is designed to reduce destructive aliasing, achieves only 73% correct prediction rate.

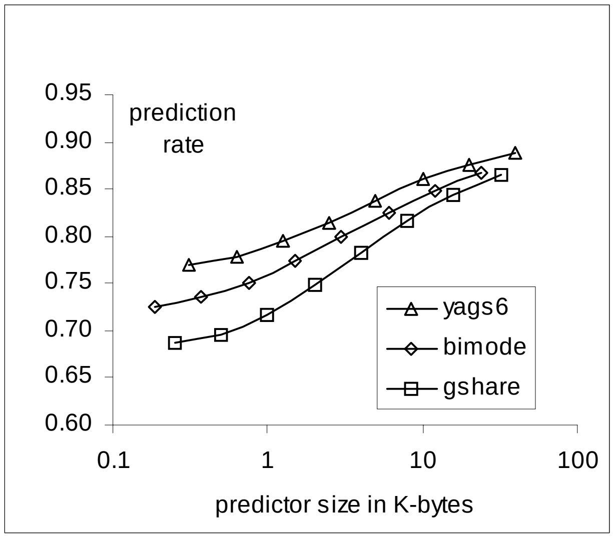

Figure 8. Predicting GO.

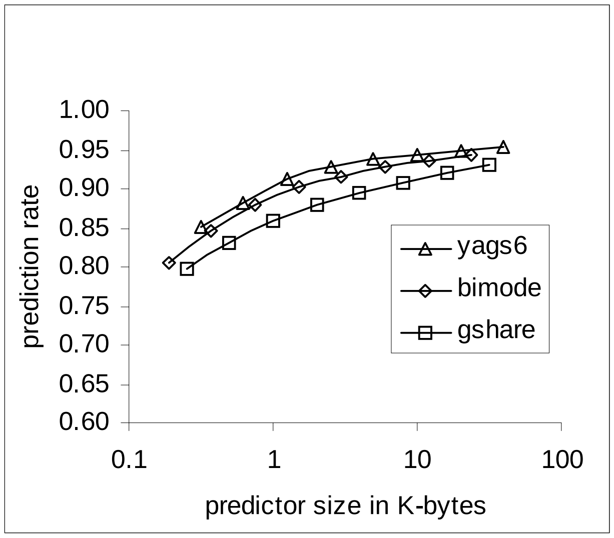

Figure 9. Predicting GCC.

4.3 Set Associativity in the Direction Caches

When increasing the size of the PHT, we increase the size of the history register to better exploit correlation between branches. However, if the direction caches are made two way set-associative, not all the bits in the history register are used to index into the direction caches. In fact, one less bit is used than if the direction caches were direct-mapped. This loss of correlation has a negative effect on the prediction rate. In the present YAGS scheme, the amount of remaining aliasing is so little that the advantage gained by making the direction PHT set-associative is offset by the loss of correlation. In order to maintain the same level of correlation, one bit of history is used as a tag in addition to the usual tag.

Figure 10 shows the prediction rate of a 6 bit tag YAGS vs.

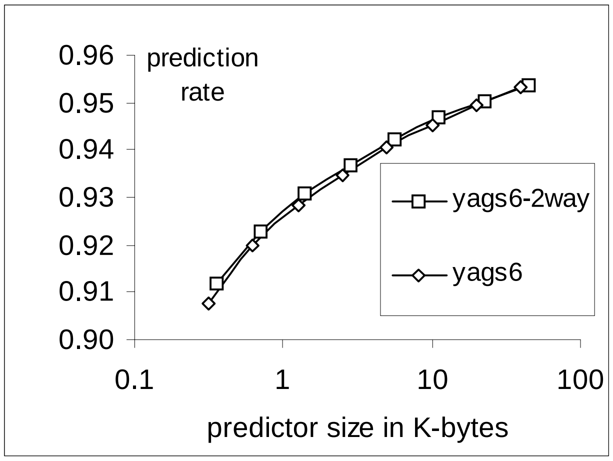

Figure 10. 6 bit tags vs. 2-way set associative.

the same predictor with a 2 way set associative cache. The extra bits that are used by the two way set-associative are the LRU bits and the extra tag bit which is taken from the history register. As expected, the two way set-associative version is able to reduce the aliasing in the direction caches. The small difference between the schemes is due to lack of aliasing in the direction caches.

4.4 Context Switching

Future high-performance microprocessors will use larger branch prediction schemes — a trend that is very likely to continue in the near future. Ideally, the prediction rate should improve in proportion to the amount of hardware put into the scheme. However, a pitfall of large predictors is the time it takes them to reach peak performance from a cold start. In the presence of intensive context switching the warm-up time of the branch prediction scheme can have a significant influence on the misprediction rate. Furthermore, some complex schemes might end up achieving less accurate predictions than a less sophisticated scheme, due to long warm-up times. It was shown that a hybrid predictor (first proposed in [1]) composed of gshare and the bimodal predictor has good performance in the presence of a context switch [9]. This is due to a short warm up time of the bimodal component. Each branch is mapped to only one entry in the PHT of the bimodal scheme. Therefore, it takes only few executions of a branch for its respective entry to reflect the information stored the branch. On the other hand, the gshare scheme has to execute a branch several times for each history instance for it to warm up. The potentially large number of history instances (i.e., $2^{\text{history length}}$) will result in a very long warm-up time and that in turn will cause a degradation in performance in the presence of context switches. The same phenomenon is observed in the skewed predictor.

However, one would expect the bi-mode predictor and YAGS to be more tolerant of context switches. Most of the information in the "not taken" direction PHT of the bi-mode predictor is "not taken." Once the choice PHT points to the "not taken" direction PHT the probability of a "taken" prediction is very small. Thus only a few executions of each branch are needed to warm up the choice PHT (it is essentially the bimodal predictor). After that, it will take more executions to warm up the branch's history instances which do not comply with the branch bias. But for the most part, the predictor will perform as well as the bimodal. The same phenomenon occurs in YAGS. This time it is due to the tags. There is a low probability that the tags will match after a context switch. Therefore, until some tags match, the choice PHT (which is, in fact, the bimodal) will serve as the predictor.

In a sense, YAGS and the bi-mode predictors are hybrid predictors which combine the gshare scheme with the simple bimodal predictor. In the presence of a context switch, they should exhibit the short warm up time of the bimodal predictor. (Similar behavior is seen in the agree predictor.)

Figure 11 shows the performance of the schemes tested in the presence of context switches. As expected, YAGS and the bi-mode predictor perform much better than gshare and the skew predictor because of their short warm-up times. The differences between the performance of the different methods is much more pronounced in the presence of context switches. The gshare scheme would converge with the others only if the PHT were large enough to accommodate most of the branch instances from all the SPEC95 benchmarks. Without context switches, the schemes would converge if the gshare PHT were big enough to accommodate the benchmark with the largest branch signature.

The gshare scheme does not perform as well as the others. This is because of its long warm-up time, as discussed above.

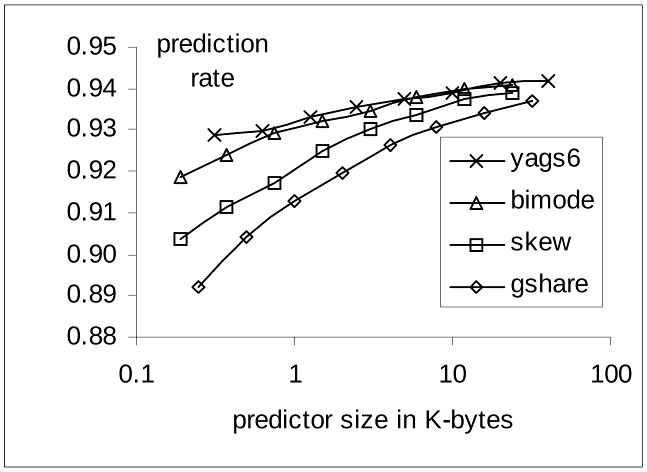

Figure 11. Predicting in the presence of context switches.

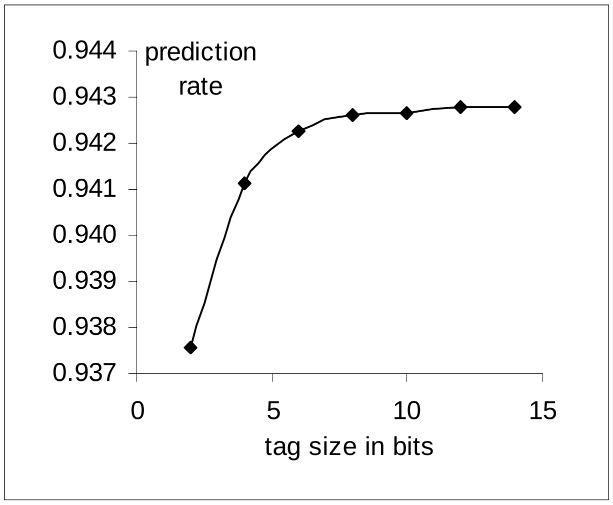

Figure 12. Tag sizes for SPEC95.

The difference between the performance of YAGS and that of the bi-mode scheme is very small. Only for very small predictor size is the difference significant. It might be that YAGS would do better in the presence of a context switch if a larger tag size were used.

4.5 Design Space

The YAGS version shown so far has a 6 bit tag and the direction caches are each half the size of the choice PHT. This is somewhat arbitrary. How big do the tags need to be to identify the branch in most cases? Figure 12 shows the prediction rate as a function of the tag size for SPEC95. The size of the choice PHT is 0.25KB (1024 entries), each direction cache has 512 entries and its size varies according to the size of the tag. According to figure 12, there is no reason to increase the size of the tag beyond 8 bits — prediction improvement is almost zero. There may be no reason to increase the size of the tag from 6 to 8 bits since the prediction improvement is very small and may not justify the increase in the predictor size. Figure 13 shows the prediction rate as a function of tag size for the go benchmark only. The difference between the prediction rate for a 6 bit tag and 8 bit tag is more noticeable for the go benchmark than for SPEC95 in general. As mentioned before, the go benchmark has a large branch signature and can benefit from an increase in tag size.

Figures 14 and 15 shown the prediction rate vs. predictor size for the SPEC95 and go benchmark respectively. On average for SPEC95, increasing the tag from 6 bits to 8 bits

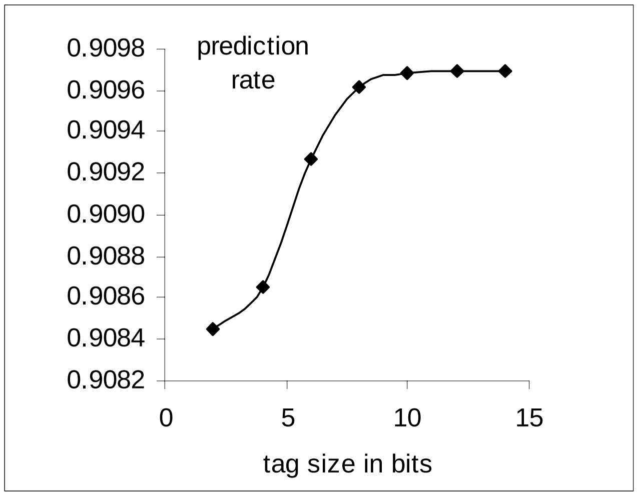

Figure 13. Tag sizes for GO.

does not result in better predictions (figure 14). On the other hand, it does improve the prediction rate for the go benchmark (figure 15). The prediction rate improvement is minimal and almost negligible. Even increasing the size of the tag to 32 bits does not result in a better prediction rate, but it increases the size of the predictor considerably.

By reducing the amount of unnecessary information stored in the direction caches, we are able to reduce the number of entries in the direction caches and to make the introduction of tags cost-effective. Figure 16 gives some insight as to

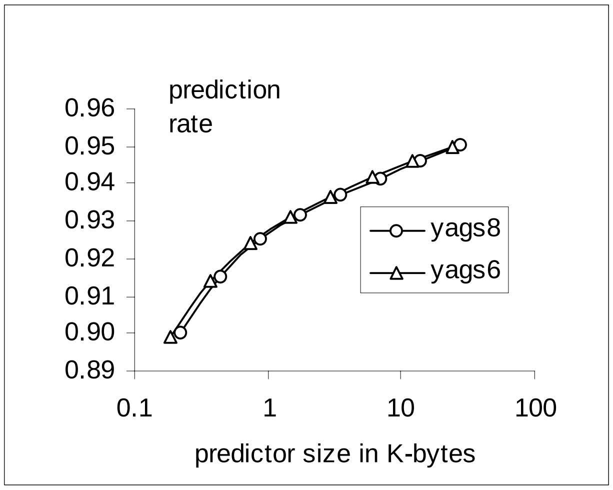

Figure 14. Predictor size for SPEC95.

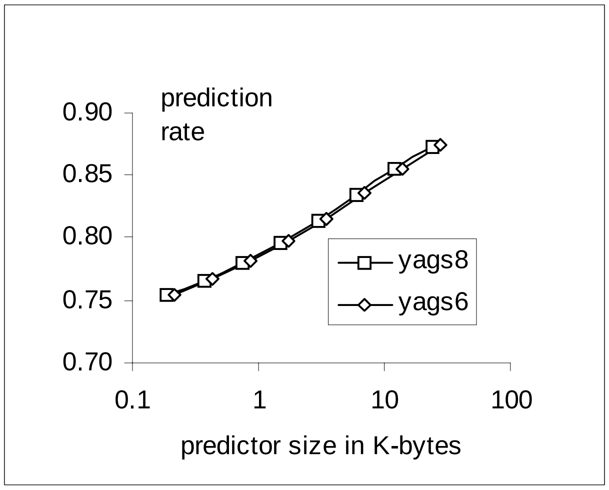

Figure 15. Predictor size for GO.

how small the direction caches can be with respect to the choice PHT. Figure 16 shows the prediction rate vs. predictor size for three versions of YAGS. The direction caches in the first version are each half the size of the choice PHT. In the second version, they are one quarter the size of the choice PHT, and in the third are one eighth of the size. All versions use a six bit tag

Figure 16 shows that for small predictor sizes the 0.125 version is best, while for large predictor sizes, the 0.5 version is best. For small predictor sizes, most of the resources should be allocated to the choice PHT, ensuring that the predictor will predict at least as well as a simple

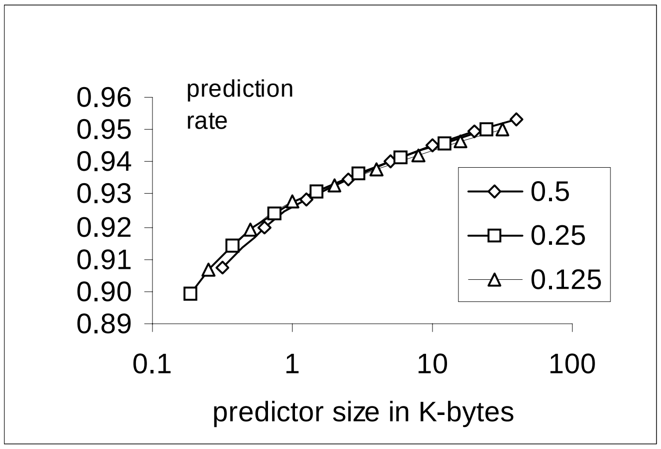

Figure 16. Direction cache size.

bimodal predictor. When the amount of resources increases, there is much less aliasing in the choice PHT and resources can be freed to handle the cases where a branch does not agree with its bias (i.e. larger direction caches). Thus the size of the direction caches should be tuned according to the overall size of the predictor.

5. Summary

We introduced YAGS, a two level global branch prediction scheme which tries to eliminate aliasing in the PHT by combining the advantages of previous schemes. YAGS performs as well as all other schemes tested. In many cases it was considerably better. YAGS and the bi-mode predictors perform well in context switches.

Some work was done to investigate the design space. Increasing the size of the tags only improves performance up to a point. After that, increasing the tag size will degrade performance, and the marginally better prediction rate does not justify the resources taken up by the larger tag. We have found that the size of the direction caches should be tuned to the size of the predictor.

We believe the potential of YAGS is greater than what we were able to demonstrate in this paper. In all experiments conducted for this paper, the size of the history register was dictated by the amount of resources allocated for the predictor. For example, in a 1KB gshare, there are 4KB entries and therefore the size the history register was forced to be 12 bits. The closest bi-mode predictor in size which was tested is a 0.75KB predictor, from which only 0.25KB (1K entries) were dedicated to each direction PHT. This forced this instance of the bi-mode predictor to use only a 10 bit history register. As a result, the bi-mode although reducing the aliasing in the PHT, has reduced correlation information for use in the prediction, compared to a similar sized gshare. This phenomena holds true for the YAGS predictor as well, since the size of the direction caches is reduced even further than in the bi-mode predictor and as a result the size the history register (and therefore the correlation information) was reduced. An example is the 1.25KB YAGS where 0.25KB are dedicated to the choice PHT. Each direction cache takes 0.5KB and has 64 entries, i.e., the history register is only 6 bits.

In figure 16, whenever the size of the direction caches was decreased by half, the size of the history register was decreased by one bit and therefore correlation information was lost. A better experiment would decrease the relative size of the direction caches while adding history bits as tags. Making the direction caches 2 way set associative hardly improved the prediction. This led us to believe that the aliasing problem in the direction PHT was almost completely solved. Therefore, decreasing the size of the direction caches degraded the performance because of the reduction in correlation information, and not necessarily because of increased aliasing.

We hypothesize that an improved YAGS would have much smaller direction caches with more history bits in the tags to preserve or increase the correlation information for use in prediction. Of course history bits can be tagged in every predictor scheme but the overhead in YAGS would be significantly smaller than all the other schemes.

Finally, the basic idea behind YAGS could be combined with other of the schemes, particularly the filter mechanism. An enhancement that might be tried is add a small cache to capture the instances filtered out of the PHT which do not agree with the bias bit.

Acknowledgments. This work was supported in part by DAPRA contract DABT63-97-C-0047. The authors would also like to thank Elly Z. Winner and C.-C. Lee for their help.

References

[1] S. McFarling. *Combining Branch Predictors*. Technical Report TN-36, Digital Western Research Laboratory, June 1993.

[2] A. Talcott, M. Nemirovsky, and R. Wodd. The Influence of Branch Prediction Table Interference on Branch Prediction Scheme Performance. *Proc. 3rd Ann. Int. Conf. on Parallel Architectures and Compilation Techniques,* 1995.

[3] C. Young, N. Gloy, and M. Smith. A comparative Analysis of Schemes for Correlated Branch Prediction. *Proc. 22nd Ann. Int. Symp. on Computer Architecture*, June 1995

[4] S. Sechrest, C.-C. Lee, and T. Mudge. Correlation and Aliasing in Dynamic Branch Predictors. *Proc. 23rd Ann. Int. Symp. on Computer Architecture*, May 1996.

[5] C. Young, N. Gloy, B. Chen, and M. Smith. An Analysis of Dynamic Branch Prediction Schemes on System Workloads. *Proc. 23rd Ann. Int. Symp. on Computer Architecture*, May 1996.

[6] P. Michaud, A. Seznec, and R. Uhlig. Trading Conflict and Capacity Aliasing in Conditional Branch Predictors. *Proc. 24th Ann. Int. Symp. on Computer Architecture*, May 1997.

[7] E. Sprangle, R. Chappell, M. Alsup, and Y. Patt, The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference. *Proc. 24th Ann. Int. Symp. on Computer Architecture*, May 1997.

[8] C.-C. Lee, I.-C. Chen, and T. Mudge. The Bi-Mode Branch Predictor. *Proc. MICRO 30,* Dec. 1997.

[9] 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.

[10] T.-Y. Yeh and Y. Patt. Two-level Adaptive Branch Prediction. *Proc 24th ACM/IEEE Int. Symp. on Microarchitecture*, Nov. 1991.

[11] T.-Y. Yeh and Y. Patt. A Comparison of Dynamic Branch Predictors that us Two Level of Branch History. *Proc. 20th Ann. Int. Symp. on Computer Architecture*, May 1993.