A Comprehensive Study of Dynamic Global History Branch Prediction

Pierre Michaud, André Seznec

INRIA

A Comprehensive Study of Dynamic Global History Branch Prediction

Pierre Michaud — André Seznec

N° 4219

Juin 2001

THÈME 1

Rapport de recherche

A Comprehensive Study of Dynamic Global History Branch Prediction

Pierre Michaud , André Seznec

Thème 1 — Réseaux et systèmes
Projet CAPS

Abstract: This report recapitulates and analyzes the most significant results on global history branch prediction during the past decade. Most global history branch predictors (GHBP) are implemented using one or several two-bit counter tables. The global branch history, i.e., the outcomes of the most recently encountered branches, is combined with the branch address to index those tables. Previous studies addressed several questions : Why use two-bit counters ? What limit for the prediction accuracy of a GHBP ? Which hardware budget is needed to come close to this limit ? How to better utilize a limited hardware budget ? This study revisits these questions. First we show that a two-bit counter is close to an optimal predictor, while allowing simple hardware implementations. Then we use a PPM algorithm as a vehicle to understand practical prediction accuracy limits for most GHBPs. Using PPM, we show that a long global history can potentially improve the prediction accuracy, even though a part of this potential is offset by the impact of cold-start misses. We observe that programs with branches that are difficult to predict are likely to have their working set increase very fast with the global history length. These programs experience a lot of cold-start misses and, also, require huge branch prediction tables to come close to PPM limit. Then we study "dealiased" GHBPs, which are approximations of PPM, given a limited hardware budget. We isolate the few dealiasing primitives constituting the "active principles" of dealiased GHBPs previously published. We study these primitives and show that some of them are actually similar. We show that simple combinations of these primitives are able to exploit long global histories for realistic hardware budgets. Finally, we conduct an experimental study on the perceptron branch predictor, which, unlike previously published GHBPs, does not derive from BPPM. Although the perceptron alone is generally not as accurate as a well-tuned "classical" dealiased GHBP, it sometimes succeeds where classical GHBPs fail. This suggests that, by using the perceptron in a hybrid predictor, it might be possible to overcome PPM limitations.

Key-words: global history branch predictor, PPM, dealiasing primitives, perceptron

Une étude de la prédiction de branchements par historique global

Résumé :   Ce rapport récapitule et analyse les résultats les plus importants de ces dix dernières années sur la prédiction de branchements par historique global. La plupart des prédicteurs de branchements par historique global (PBHG) utilisent une ou plusieurs tables de compteurs deux bits accédées par l'adresse du branchement et l'historique global constitué des directions des branchements les plus récents. Les études précédentes ont soulevé plusieurs questions : Pourquoi utiliser des compteurs deux bits ? Quelle est la valeur limite du taux de mauvaises prédictions ? Quel budget matériel est nécessaire pour approcher cette limite ? Comment faire meilleur usage d'un budget matériel donné ? L'étude présente se focalise sur ces questions. D'abord, nous montrons que le compteur deux bits est proche d'un prédicteur optimal, tout en permettant une mise en oeuvre matérielle simple. Puis nous utilisons l'algorithme PPM pour étudier les limites pratiques des PBHG usuels. Nous montrons qu'un historique global long, potentiellement, diminue le taux de mauvaises prédictions, mais qu'une partie de ce potentiel est annulée par les défauts de démarrage à froid. Nous observons que les programmes comportant des branchements difficiles à prédire ont un ensemble de travail qui croît rapidement avec la longueur d'historique global. Ces programmes subissent un grand nombre de défauts de démarrage à froid et nécessitent de très grandes tables de prédiction. Ensuite, nous étudions les PBHG à budget matériel limité. Nous identifions les quelques primitives anti-interférence caractérisant les principaux PBHG connus. Nous étudions ces primitives et montrons que certaines combinaisons de ces primitives permettent d'exploiter un historique global long, étant donné un budget matériel raisonnable. Enfin, nous menons une étude expérimentale du perceptron utilisé comme PBHG. À la différence des PBHG précédents, le perceptron n'est pas une approximation de PPM. Notre étude montre que le perceptron, seul, n'est en général pas aussi performant qu'un PBHG classique. Cependant, le perceptron est capable sur certains programmes de surpasser un PBHG classique. Cela suggère la possibilité, en utilisant le perceptron dans un prédicteur hybride, d'obtenir un taux de mauvaises prédictions inférieur à celui de PPM.

Mots-clés :   prédicteur de branchements par historique global, PPM, primitives anti-interférence, perceptron

Contents

1   Introduction

Branch prediction is a "keystone" of high performance in pipelined out-of-order processors. Hardware branch predictors have been studied a lot in the past decade since the introduction of two-level branch predictors by Yeh and Patt [39]. Results of these studies are disseminated in many publications. This study is a recapitulation of some the most significant results published in the past decade.

We have chosen to focus our attention on a particular type of branch predictors : dynamic *global history* branch predictors (GHBP). These predictors, which are among the most accurate, are used in several commercial processors.

Global history branch predictors were introduced in [40] and [27]. A GHBP is a *two-level* predictor [40] which first history level is formed by a single shift register recording the outcomes of branches encountered the most recently. Such predictor exploits correlations between branches in programs.

Figure 1 depicts a typical branch prediction pipeline stage based on a GHBP. The global branch history is maintained with a shift register. The branch address and global history bits are hashed together to index a table of two-bit counters. Each two-bit counter is a compact history of the outcomes of all previous (address, history) pairs that were mapped onto that particular counter (ideally, each pair is mapped onto a distinct counter). A prediction is obtained from a two-bit counter by reading its most significant bit.

In a GHBP, the global history can be, and should be, updated speculatively with the predicted branch outcome [41, 14]. As long as predictions are correct, the global history is correct. When a misprediction occurs, no matter what happens to the global history : after detecting the misprediction, subsequent instructions are flushed and the global branch history is restored from a copy recorded in the checkpoint [15, 19].

The organization of the present study is as follows. In Section 2, we justify the use of two-bit counters as the basis of most GHBPs. We also introduce *partial update*, which plays an important role in dealiased branch predictors (cf. Sections 5 and 6). Section 3 describes suppositions we made for evaluating predictors. In Section 4, we study *branch prediction by partial matching* (BPPM), an algorithm originally used for data compression and which was adapted to branch prediction by Chen, Coffey and Mudge [4]. We use BPPM to characterize the working set of the benchmarks used in the remaining of this study. Real branch predictors, of course, have to deal with limited storage capacities, and the working set characteristics determine the storage capacity required to store branch prediction information. A limited storage capacity for branch prediction information induces aliasing in predictor tables. So called dealiased predictors try, using *dealiasing primitives*, to overcome aliasing effects and approximate the prediction accuracy of BPPM, BPPM being a limit for these predictors. Section 5 introduces the aliasing problem, and describes six dealiasing primitives extracted from predictors previously proposed in the literature. In Section 6, we perform an experimental study of these dealiasing primitives. Section 7 is an experimental evaluation of the perceptron, which was recently proposed as a branch predictor in [18]. Unlike most GHBPs introduced previously, the perceptron is not derived from BPPM. We
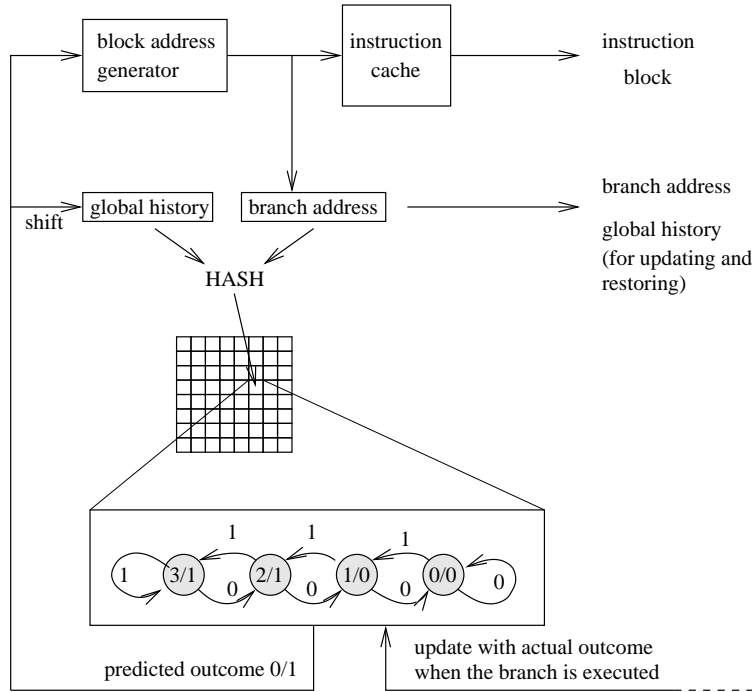
Figure 1: Example of global history branch predictor.

show that, for some benchmarks, the perceptron is able to succeed where classical predictors fail.

2 Two-Bit Counters

Most GHBPs introduced previously are implemented with two-bit counter tables, as illustrated on Figure 1. This section presents a few reasons justifying the use of two-bit counters. In particular, we show that a two-bit counter is close to an optimal predictor for "stochastic" branches, while allowing simple implementations for prediction tables.

2.1 The Two-Bit Counter as an Approximation to an Optimal Predictor

Two-bit up-down saturating counters are the basis of most branch predictors [32, 40]. An exhaustive comparison of all four-state automatons [26] showed experimentally that the
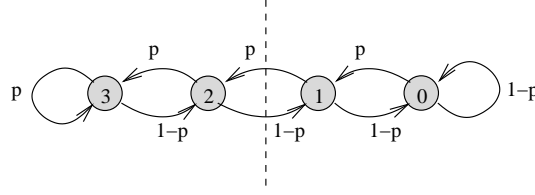
Figure 2: Two-bit counter as a Markov process.

two-bit up-down saturating counter is one of the best four-state automatons for branch prediction, if not the best.

In this section, we show with a simple model that a two-bit up-down saturating counter is close to an optimal predictor when trying to predict the outcomes of Bernoulli trials which bias is not known *a priori*.

We focus on a branch which we assume can be modeled by a Bernoulli process, with a probability p that the branch is taken and a probability $1 - p$ that it is not taken. A two-bit counter used to predict that branch can be modeled as a Markov process, as depicted on Figure 2.

**Initial behavior of the two bit counter.** On the first occurrence of the branch, we have no past history. In particular, we do not know the branch bias. If we make a random prediction, the probability m_1 to have a misprediction on the first occurrence is $m_1 = 50\%$. Then the counter is initialized according to this first outcome. This can be a "strong" initialization (state 3 if the first outcome is *taken*, state 0 otherwise) or a "weak" initialization (state 2 if the first outcome is *taken*, state 1 otherwise). A simple calculus shows that probabilities m_2 and m_3 to have a misprediction on the second and third occurrences, respectively, are equal, and do not depend on the "strength" of the initialization

$$m_2 = m_3 = 2p(1 - p) \tag{1}$$

This means that the "strength" of the initialization has no impact on the quality of the prediction. The only important thing is to initialize the counter according to the first outcome.

It should be noted that the misprediction probability expressed in Formula 1 is also the misprediction probability of a 1-bit counter which predicts the same outcome as the previous occurrence.

Asymptotic behavior of the two-bit counter Let P_n be the probability to be in state 3 or 2 after the nth occurrence of the branch, and $1 - P_n$ the probability to be in states 0 or 1. Let us consider two consecutive occurrences of the branch, and the counter states before

and after these two occurrences. Crossing the dashed line (cf. Figure 2) implies that the two consecutive occurrences have the same outcome. Hence,

$$
\begin{aligned}
P_{n+2} &= p^2(1 - P_n) + (1 - (1-p)^2)P_n \\
&= p^2 + 2p(1-p)P_n
\end{aligned}
$$

As $2p(1-p) \leq \frac{1}{2}$, sequence P_n converges **quickly** toward P_∞ defined as

$$
P_\infty = p^2 + 2p(1-p)P_\infty
$$

$$
P_\infty = \frac{p^2}{1 - 2p(1-p)} \tag{2}
$$

Practically, for $n > 2$, P_n is close to P_∞. The misprediction probability m_∞ is equal to $P_\infty(1-p) + (1 - P_\infty)p$

$$
m_\infty = \frac{p(1-p)}{1 - 2p(1-p)} \tag{3}
$$

Of course, the prediction of the two-bit counter is not optimal. An optimal [1] predictor would predict the branch always in the same direction. For example if the branch bias is *not taken*, the misprediction probability of the optimal predictor is $m_{opt} = p$.

Figure 3 shows the *asymptotic* and *initial* misprediction probability of the two bit counter for $p \in [0..1]$, compared with the optimal predictor.

On the second and third occurrence of the branch, the prediction is rather far from the optimal, as shown on Figure 3. For $p < 0.1$ or $p > 0.9$, the probability of misprediction on the second and third occurrence is twice the misprediction probability of the optimal predictor.

Once asymptotic probabilities are reached, the two-bit counter is very close to the optimal predictor for $p < 0.1$ or $p > 0.9$. Considering m_∞/m_{opt}, the worst case for the two-bit counter is for p around 0.3 or 0.7. In this case, the misprediction probability is about 20% higher than the optimal value.

Instead of two-bit counters, we could consider three-bit counters. A three-bit counter, for a "stationary" branch, is closer to the optimal predictor. When not considering silicon budget limitations, from our experimentations, three-bit counters are slightly better than two-bit counters, especially when the global history is short. However, when considering hardware constraints, three-bit counters consume 50 % more space than two-bit counters and are more difficult to update.

---

[1]This optimal predictor is theoretical because it knows the branch bias *a priori*.
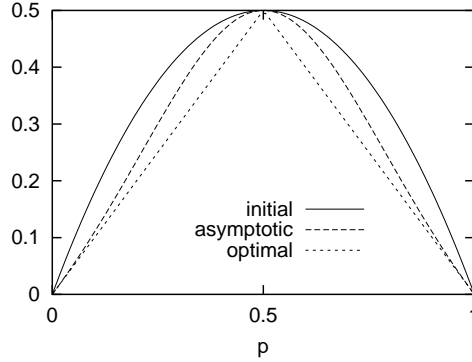
Figure 3: Initial and asymptotic misprediction probability of the two-bit counter as a function of p (probability that the branch is taken), compared with the optimal predictor.

2.2   Splitting Two-Bit Counters

As shown on Figure 1, the two-bit counter table is simultaneously read at the prediction pipeline stage and updated at the execution pipeline stage. Seemingly, the table ought to be dual-ported. However, it is possible to use single-ported tables by taking advantage of the fact that, when the two-bit counter prediction is correct, only the least significant bit needs to be updated : we set it to the branch outcome, that is, we strengthen the counter.

The two-bit counter table can be split in two tables T1 and T0, as shown on Figure 4. The most significant counter bit is stored in T1 and the least significant bit is stored in T0. As long as predictions are correct, the most significant counter bit b_1, which gives the prediction, does not change. The least significant bit b_0 is updated by being set to the branch outcome. Upon a misprediction, both tables are updated : we copy b_0 from T0 to T1 and we complement b_0 in $T0$. There may be a one cycle penalty for updating $T1$, but this penalty may be overlapped with penalties already existing (pipeline flush, fetch address and global history restoration ...).

2.3   Partial Update

This technique of split counters is possible only for certain branch predictors : predictors using a *partial update*. This type of update was identified in [24] as a way to decrease the impact of aliasing in predictors consisting of several sub-predictors :

> *when the overall prediction is correct, update (i.e., reinforce) only the sub-predictors which gave a correct prediction*

It should be noted that this only defines a family of update methods. There may be several partial update methods for a given branch predictor. According to this definition, single-
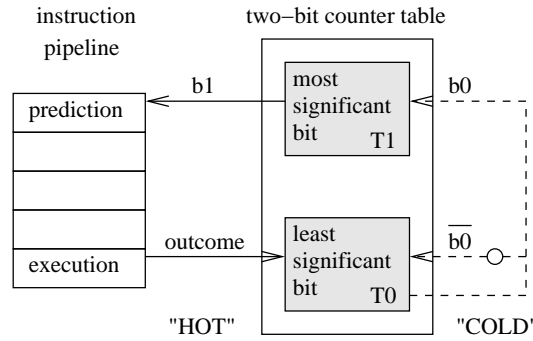
Figure 4: Split two-bit counter table. The "cold" update path is used only upon a misprediction.

table predictors, like the one described on Figure 1, use a partial update. A consequence of partial update is that **the most significant bit of a two-bit counter changes only upon a misprediction**.

3   Experimental Framework

The remaining of this study is mainly an analysis of experimental results obtained with simulations. This section describes our experimental set-up and suppositions we made for evaluating predictors.

Sequential-update model.   In most branch prediction studies considering the branch predictor as a stand-alone mechanism, it is assumed that the predictor is updated with the branch outcome just after the branch is predicted, and just before predicting the next branch. Moreover, it is assumed that this update is done in sequential order. Throughout this study, we used this sequential-update model.

However, in a pipelined processor, there is a delay for updating the predictor. Moreover, in out-of-order processors, the update may be done in an order different from the sequential order.

If the predictor table is updated at instruction retirement, i.e., non-speculatively and in sequential order, then the sequential-update model is an exact model. Predictions obtained between the prediction and the execution of a mispredicted branch may differ from the predictions obtained with the sequential-update model. But this corresponds to instructions which will be flushed from the processor after the misprediction is detected. After resuming from the branch misprediction, the branch predictor is in a state coherent with the sequential-update model.

If the predictor table is updated at execution instead of retirement, the branch predictor may be in a state non-coherent with the sequential-update model. This concerns out-of-order processors. Previous studies have shown that speculatively updating a branch predictor may have a beneficial effect [19], due to the reuse of control-independent work [33].

In this study, we did not evaluate the possible impact of updating the predictor out of the sequential order.

Branch predictors hardware cost.   For comparing branch predictors, we compare their prediction accuracy for a fixed hardware cost. The hardware cost is not precisely defined : it may be the silicon area, constraints on the branch prediction bandwidth or latency, power consumption, a combination of all these constraints ... This depends on the context. In this study, we have chosen, as in most previous studies, to measure the cost of a branch predictor as the total memory capacity required for branch prediction tables (i.e., the total number of bits).

For some experiments, we consider large hardware budgets that may exceed 100 KBytes. In general, this is for providing insight on the impact of hardware budget limitation on prediction accuracy. However, we must point out that the size of branch predictors implemented in commercial processors increases at each technological generation.

The current trend is to have multiple levels of branch predictors, accessed at different pipeline stages [17]. This is the case, for example on the Alpha 21264 [21], which has two branch prediction levels: a level-zero (L0) and a level-one (L1) predictor. On the Alpha 21264, the L0 predictor is a next-line predictor [3], accessed in less than one cycle. The L1 predictor is a GHBP, more accurate than the L0 predictor, but which takes one extra cycle to access. Once the L1 prediction is known, it is compared with the L0 prediction. If predictions differ, the instruction flow is re-directed using the L1 prediction, which incurs a one-cycle penalty.

It is likely that future processors will also feature a L2 predictor, pipelined over several cycles, so that, when the L1 prediction is wrong but the L2 prediction is correct, we pay the cost of a partial pipeline flush instead of a full pipeline flush. A L2 predictor will be typically much larger than predictors implemented currently.

However in the remaining, we consider GHBPs without distinguishing between L1 and L2 predictors.

Experimental set-up   All experimental results in this study were obtained with a trace-driven simulator. We used twelve benchmarks. Eight of our benchmarks are the IBS traces *groff*, *gs*, *mpeg_play*, *nroff*, *real_gcc*, *sdet*, *verilog* and *video_play* (cf. [38]). An interesting aspect of these traces is that they contain both user and system activity. The four other benchmarks are taken from the SPEC CPU suite [34]. We took *go* from the SPEC CPU95 suite. This benchmarks was chosen because it is notoriously difficult to predict. Although not representative of an "average" application, it is interesting as an extreme case. The three other SPEC benchmarks are *mcf*, *twolf* and *gap* from the more recent SPEC CPU2000 suite.

For SPEC benchmarks simulations, we used Simplescalar tools [1]. The three CPU2000 programs were compiled for simplescalar portable ISA with *gcc* 2.7.2.3 "-O3". Applications were run to completion using the *test* inputs. Some statistics on the benchmarks are reported in Table 1.

As said previously, all subsequent simulation results are based on a sequential-update model. In this study, branch prediction accuracy is represented either by the branch misprediction ratio (ratio of the number of mispredictions over the number of dynamic branches) or by the misprediction interval, defined as the inverse of the misprediction ratio. When comparing predictors, what matters is not the absolute difference, but the relative difference. Hence these two representations are equivalent, and the choice to use one instead of the other is just a question of aesthetic.

4 Branch Prediction by Partial Matching (BPPM)

The Prediction by Partial Matching (PPM) algorithm was introduced by Cleary and Witten [5] as part of a data compression algorithm. It was shown to provide very good compression ratios on English texts. More recently, Chen, Coffey and Mudge [4] have shown that this technique can be used for branch prediction.

Actually, traces generated by programs control flow have characteristics in common with English texts. In particular, paths in the control flow can be viewed as strings. Just like strings, some paths have a greater probability of occurrence than others. PPM is a useful theoretical tool for studying the limits of usual GHBPs, which approximate a PPM algorithm, as noted in [4].

Branch Prediction by Partial Matching (BPPM) uses a set of $h_{max} + 1$ Markov predictors with orders ranging from h_{max} to 0. Each static branch B in the program has its own BPPM predictor, and the string searched is the global branch history H_h of length $h \leq h_{max}$ for dynamic instances of that branch. For each (B, H_h) pair, we record the history of branch outcomes previously encountered in that context.

For making a prediction, we search the Markov predictor of order h_{max}. If there is a match, which means $(B, H_{h_{max}})$ was encountered at least once in the past, the history of previous outcomes associated with $(B, H_{h_{max}})$ is used to make a prediction. If there is a miss, the Markov predictor of order $h_{max} - 1$ is searched. If there is also a miss, the Markov predictor of order $h_{max} - 2$ is searched, and so on... If there is a miss in the Markov predictor of order 0, this is the first time branch B is encountered, and we make a static prediction (for example we predict that the branch is taken).

There are possible variations of BPPM. In the BPPM predictor proposed in [4], the history of previous outcomes is simply a count of 1's and 0's. *Adaptive* PPM predictors studied in [11], on the other hand, give more weight to recent outcomes. It was shown in [11] that *adaptive* BPPM gives generally slightly better results because of branches with a non-stationary behavior.

In the BPPM predictor we simulated, the history of previous outcomes is maintained with a two-bit counter [2].

Another possible variation concerns the way the BPPM predictor is updated. In [4], *update exclusion* was used : if the longest match is in the Markov predictor of order h, we do not update the counters associated with (B, H_i) pairs such that $i < h$. Update exclusion generally gives better results when compressing English texts. However, on BPPM predictors, we found a full update to give slightly better results.

Simulation. There are two possible simulation algorithms for BPPM. A first possibility is to simulate $h_{max} + 1$ tables of unbounded size, each table corresponding to a different history length, i.e., a different Markov predictor order. Entries in the tables are tagged with (B, H_h) pairs. A second possibility is to simulate a forest of binary trees, as in [11]. There is a binary tree for each static branch encountered in the instruction trace. We have chosen the latter solution.

It should be noted that, with a full update, a BPPM predictor of order h is included in all BPPM predictors of order $i > h$. Hence we were able to collect statistics for all the BPPM predictors of order h ranging from 0 to h_{max} with a single simulation, by going from the root of the tree down to the maximum history length (which is the depth of the tree).

For simulations on the IBS benchmarks, we simulate global history lengths up to $h_{max} = 64$. For IBS *real_gcc*, we provide results only for half of the instruction trace (cf. Table 1, *half real_gcc*) because the memory capacity of our machine was exceeded. For the four SPEC benchmarks, we set $h_{max} = 32$ also because of memory limitations.

4.1   Characterization of BPPM Working Set

The solid curve on Figures 12 and 13 represents the number W_h of distinct (B, H_h) pairs that were encountered at least once, divided by the number of dynamic conditional branches. The dashed curve shows the number of distinct (B, H_h) pairs that were encountered at least twice. Note that the solid curve can also be viewed as a miss ratio.

We observe that the working set size increases much slower than 2^h. Actually, for some benchmarks, the working set size does not grow exponentially with h, but rather linearly.

A possible explanation could be that a branch can be executed only when certain branches take a certain direction. For example, consider a branch B in the THEN or ELSE path of an IF statement : branch B can be executed only when the IF branch takes a certain direction. Another example : if branch B is after a loop, branch B can be executed only after the loop branch is not taken. However, we think this cannot explain the fact that the growth is much slower than 2^h. Let us consider the graph of static conditional branches. A directed edge between branches A and B means that B may be executed after A. If we neglect the effect of indirect jumps (like function returns), each node has 2 output edges, corresponding to *taken* and *not-taken* (with indirect jumps, the average number of output edges is greater than

---

[2]BPPM using three-bit counters gives slightly better results, but using two-bit counters eases the comparison with hardware predictors of following sections, which are based on two-bit counters

2). In any directed graph, the average number of input edges equals the average number of output edges. Hence, branches have two input edges **on average**, and for high values of h, the number of paths leading to a branch is 2^h.

Actually, there are paths that will never be taken. Some conditional branches behave like unconditional branches. Roughly, 30% of dynamic branches are instances of static branches that always took the same direction during a particular program execution [10]. We can remove 30% of branches from the global history without losing path information.

Second, there are correlations between branches. This concerns IF statements, but also loop branches. For example, consider a loop with a fixed number N of iterations. After exiting the loop, we can remove the N instances of the loop branch from the global history because they carry no path information. If the number of loop iterations is variable but always greater than or equal to N, then, after exiting the loop, we can still remove N instances of the loop branch without losing path information. Even if there are x instances of a loop branch in the global history that cannot be removed because the loop is not finished, instances of the loop branch carry only $log_2 x$ bits of path information.

A global history H_h can be reduced to a history H'_k of k branches without losing path information. The number of possible global histories is 2^k, with $k < h$.

Figures 14 and 15 show the curve of W_h/W_0 on a logarithmic scale, as a function of the global history length h. For h not exceeding 10 or 15, depending on the benchmark, W_h/W_0 can be approximated by a α^h model, which corresponds to a straight line on the graph. For longer global histories, the α^h model diverges from the experimental curve.

For the IBS benchmarks, α is mostly between 1.20 and 1.25. For the SPEC benchmarks, α is higher : close to 1.4 for *go* and close to 1.3 for the 3 others. From the value of α, we can deduce k

$$\alpha^h = 2^k$$

In particular, for the IBS,

$$k = h \times \log_2(\alpha) \approx h/3$$

This corresponds typically to 30% of branches being totally biased and 60 % of remaining branches removed by correlations. This explains why the number of paths grows much slower than 2^h, however this does not explain why the growth is no longer exponential for long global history lengths. We have to consider the following two points :

- some paths have a greater probability of occurrence than others.

- the number of paths that have been taken at least once increases with the trace length n (second column of Table1), and it cannot be greater than n.

A model is developed in Appendix A for providing a better understanding of the dynamic growth of a working set. The model studies the theoretical case of a set of k-bit strings generated by a Bernoulli process.

Here we are modeling the behavior of a single "average" branch. Parameter k is the length of H'. Parameter n is the number of dynamic occurrences of this "average" branch. Parameter p is the probability that a branch in H' takes the direction opposite to its bias. Some curves obtained with the model are plotted on Figure 10. The model is able to reproduce the shape of some of the curves on Figures 12 and 13.

The model is, of course, a simplification of reality. This "average" branch is artificial, all static branches are not executed with the same frequency. Nevertheless, the model provides a support for a qualitative analysis of the experimental results of Figures 12 and 13.

In the following p represents the branch predictability. It is roughly correlated with numbers in the sixth column of Table 1. Parameter n represents the number of dynamic instances of the "average" static branch. It is correlated with the number of dynamic branches (second column of Table 1) over the number of static branches (fourth column of Table 1).

General behavior. The working set, once gathered, grows exponentially with h. However, a larger working set takes longer to gather. Hence, at a fixed time, the working set is exponential with h up to a certain h. Beyond this h, the working set is not gathered yet, and the curves looks like a polynomial which degree increases logarithmically with n.

The model defines the "useful" working set as the working set gathered until the instantaneous miss ratio becomes negligible. The model shows that the useful working set size depends on p and that, for long global histories, a small increase of p leads to a large increase of the useful working set. Practically, this means that for a high p and a long global history, there is a residual miss ratio which will take a very long time to decay.

In other words, programs with branches that are "difficult" to predict are likely to experience a lot of cold-start misses and will require large branch prediction tables.

IBS benchmarks. Benchmarks *real_gcc*, *mpeg_play* and *nroff* exhibit behaviors different from the 5 other IBS benchmarks.

The behavior of *real_gcc* (at this point of execution) is characterized by a small n value. A small n means that the working set of each static branch is growing rapidly and represents a significant fraction of n (the shape of the curve, quasi linear, is typical of this state). Hence the overall miss ratio is high.

The behavior of *nroff* is opposite to that of *real_gcc*. It is characterized by a relatively high n value and a very good branch predictability. A high n and a small p means that the working set accumulated on each static branch is small compared with n. This explains why the overall miss ratio is low.

The behavior of *mpeg_play* is also interesting. Its n value is of the same order of magnitude as that of other benchmarks like *groff*, *verilog* or *video_play*. But *mpeg_play* is characterized by a high p value : it has branches more difficult to predict than other IBS benchmarks, even considering a long global history. The model in Appendix A highlights the fact that a small increase of p increases significantly the working set accumulated at a given time (cf. curves of Figure 10 and Formula 17). The curve of *mpeg_play* is typical : for a fixed n, a

higher p is characterized by a more pronounced curvature. The result is a high global miss ratio.

SPEC benchmarks. The four SPEC benchmarks are characterized by a high n (*mcf, twolf, gap*) and a high p (*go, twolf, mcf*). This results in a miss curve with a strong curvature (Figure 13).

The high n makes the solid and dashed curves on Figure 13 diverge for a global history longer than the IBS. The high n and p also makes the α^h model diverge for a global history longer than the IBS (*mcf* and *twolf*, Figure 15).

The behavior of *gap* is characterized by a very high n and a relatively small p, hence a low miss ratio (somewhat similar to IBS *nroff*). For *go*, the high value of p, combined with a relatively small value of n, incurs a very high miss ratio.

4.2   BPPM Branch Misprediction Ratio

The solid curve on Figures 16 and 17 represents the misprediction interval of BPPM, that is, the number of dynamic conditional branches per misprediction (higher is better). The dashed curves represent the inverse of the misprediction probability on (B, H_h) pairs that were already encountered at least one time and three times respectively.

The solid curve represents a limit for predictors derived from BPPM, whereas the dashed curves represent a limit for global history branch prediction. The difference between the solid curve and the dashed curves shows the impact of misses on BPPM prediction accuracy.

For small global history lengths, the solid curve and dashed curves are very close. This is because the miss ratio for small history lengths is negligible. However, as the global history becomes longer, the number of misses becomes significant, and the prediction is often given by lower order Markov predictors.

If there were no misses, the misprediction interval would increase roughly linearly with the global history length. A significant part of this potential is offset by misses. Nevertheless, very long global histories ($h > 20$) are able to improve the prediction accuracy substantially, as was already observed in [18].

Several reasons can explain this. It was observed in [10] that correlated branches are generally close to each other in the source code of a program. Dynamic instances of correlated branches may be separated, in the instruction stream, by a function call, or by a tight loop. A long global history is needed to capture the correlation.

Self correlation is another reason. A long global history is able to capture a significant fraction of self correlations. For example, consider a loop with two conditional branches in its body, one being the loop branch. If the global history is longer than twice the number of iterations, it is able to capture the self correlation of the loop branch.

5   Aliasing, and Dealiased Global History Branch Predictors

Real branch predictors have to deal with strong hardware constraints, in particular a limited hardware budget. We must use the storage capacity available as efficiently as possible.

We may distinguish two types of predictors : single-table and multi-table predictors. Example of single-table predictors are *bimodal, gshare, gselect* [23].

Previous studies have shown that single-table predictors suffer from *aliasing* (a.k.a. interferences between branches) [42, 30, 37]. Aliasing is analogous to cache misses : it comes from conflicts, capacity constraints, and cold-start effects. Aliasing can be harmless or destructive [42]. Destructive aliasing has a detrimental effect on branch prediction accuracy.

5.1   Aliasing Model

It was shown in [24] that, in a single-table predictor, a large fraction of aliasing is conflict aliasing. This is a manifestation of the so-called birthday "paradox", which observes that the probability, in a group of 23 people, that two people share the same birthday is approximately 50% [6].

Let us consider a program with n static branches [3] and a branch predictor table with N entries. We try to find the number of branches that are mapped on a table entry. We define a *q-load* entry as an entry onto which exactly q static branches are mapped.

This problem [4] , although presented differently, is similar to the problem discussed in Appendix A.1, and we can use Formula 11. The average fraction F_q of q-load entries in the table is

$$F_q = v_{q-1}(x) - v_q(x) = e^{-x} \frac{x^q}{q!} \tag{4}$$

with $x = n/N$, i.e., the number of static branches over the number of predictor table entries. One will recognize a Poisson distribution. The average fraction f_q of static branches that are mapped on a q-load entry is

$$f_q = \frac{F_q N \times q}{n} = F_{q-1} \tag{5}$$

In particular, the fraction of static branches mapped on an aliased entry $(q > 1)$ is

$$1 - f_1 = 1 - e^{-x} \approx \frac{n}{N} \qquad \text{when } n \ll N \tag{6}$$

---

[3]For easing the discussion in Section 5.1, we consider static branches (i.e., $h = 0$), but the model applies also to (B, H_h) pairs.

[4]This problem is sometimes referred to in the literature as the *classical occupancy problem* [12]

To derive a misprediction ratio from this model, we introduce the fraction b of static branches having bias *taken*, and we assume a one-set model such that all static branches contribute equally to dynamic branches.

For small x values, most aliased entries are 2-load entries. The probability that two branches mapped on a 2-load entry have opposite biases is $2b(1-b)$. As the two branches are supposed to contribute equally to dynamic branches, the two-bit counter stored in that 2-load entry "sees" a stream of outcomes with 50% of 1's and 50% of 0's. We will assume that the probability to mispredict a branch when using this entry is 50%. Considering only mispredictions resulting from aliasing, the overall misprediction probability can be estimated as

$$m_{alias} \approx b(1-b)\frac{n}{N} \tag{7}$$

In particular when $b = 50\%$,

$$m_{alias} \approx \frac{n}{4N} \tag{8}$$

This model shows that, even when the number of entries is much larger than the number of branches in a program (in a function, in a loop ...), the impact of aliasing on prediction accuracy is still significant. For example, let us suppose a misprediction probability of 5% when there is no aliasing (intrinsic misprediction probability). If the number of table entries is 10 times the number of static branches, aliasing incurs 50% extra mispredictions according to Formula 8 (for a total $5\% + 1/(4 \times 10) = 7.5\%$) while more than 90% of the predictor table space is not used.

Of course, this model is too simple to reflect the exact behavior of programs. In particular, a one-set model with all static branches contributing equally to dynamic branches is obviously not valid for complex programs.

Multi-set model.   The program behavior could be better modeled by considering multiple static branch sets (S_i) such that $S_i \subset S_j$ for $i < j$, each set featuring n_i static branches and contributing to a fraction d_i of dynamic branches. For example, a program consisting of a double-nested loop would be modeled with two sets $S_1 \subset S_2$, S_1 featuring branches in the inner loop and S_2 featuring the whole loop nest.

5.2   Some Dealiasing Primitives

In a single-table GHBP like *gshare* [23], a lot of the available space is wasted, as shown by the previous model. This observation led to proposing ways to better utilize the predictor space and overcome aliasing effects. In a GHBP, the aliasing problem is amplified in two ways :

- The working set size increases with the history length, hence the probability of destructive aliasing. There is a history length h_{opt} beyond which, instead of improving the prediction accuracy, we degrade it. This history length h_{opt} depends on the predictor size. It depends also on the application, and it may even vary during the execution of a single application [20]. In the remaining, we will refer to this problem as the *history length dilemma.*

- As we increase the history length, we decrease the intrinsic misprediction probability (i.e., without aliasing). Consequently, the impact of aliasing becomes more important. For example, let us consider 1% of destructive aliasing. If the intrinsic misprediction probability is 10%, removing aliasing means removing 10% of mispredictions. On the other hand, if the intrinsic misprediction probability is 2%, removing aliasing means removing 50% of mispredictions.

Several "dealiased" GHBP have been proposed in the literature, among which McFarling's *bimodal/gshare* hybrid [23],, *e-gskew* and 2BC-gskew [24, 31], *bi-mode* [22], *YAGS* [7]. Other predictors have been proposed that use profile information passed to the hardware by compiler hints (see [28] for example). In this study we focus our attention on purely dynamic GHBPs. Also, we consider conditional branch predictors independently from the branch target buffer (some processors do not have any BTB). Previously proposed dealiased predictors share common characteristics :

- They use multiple tables and multiple global history lengths

- Prediction information may be redundantly replicated across tables

- They use a partial update (cf. Section 2.3)

- Tables are indexed with different hashing functions

The reason for using multiple global history lengths is to decrease cold-start and capacity aliasing effects (history length dilemma) : if an entry is aliased in a table because the working set is too large, we will take the prediction from a table with a shorter global history and hence a smaller working set. The cost of redundant information is a problem in dealiased predictors, and the goal of partial update is to minimize redundancy without impairing the prediction accuracy. However, we have seen in Section 5.1 that the space of a single table is very poorly used. Then we can afford a certain amount of redundancy if this redundancy allows to overcome aliasing effects. Often, dealiased predictors use different hashing functions to index different tables. This decreases the impact of conflict aliasing by decoupling conflicts : if a table entry is aliased because of a conflict, there is a good chance that the entries read in other tables are not aliased. It should be noted that indexing tables with different global history lengths is often sufficient for decoupling conflicts.

From dealiased predictors previously proposed, we extracted a few dealiasing primitives that are listed on Figure 5. A primitive describes a way to obtain a new predictor from input predictors. In particular, it gives a method for obtaining the overall prediction and
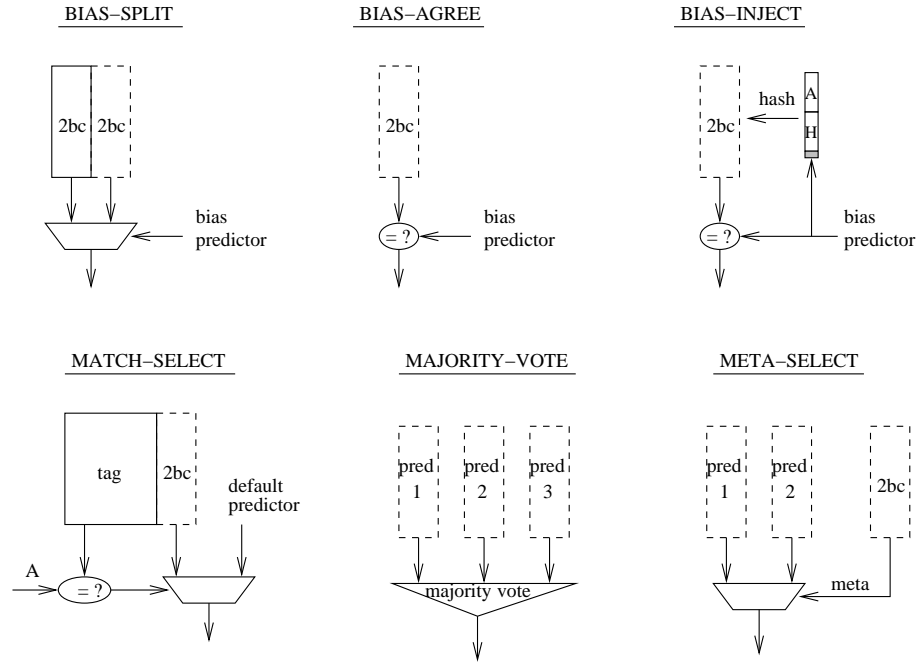
BIAS–SPLIT BIAS–AGREE BIAS–INJECT
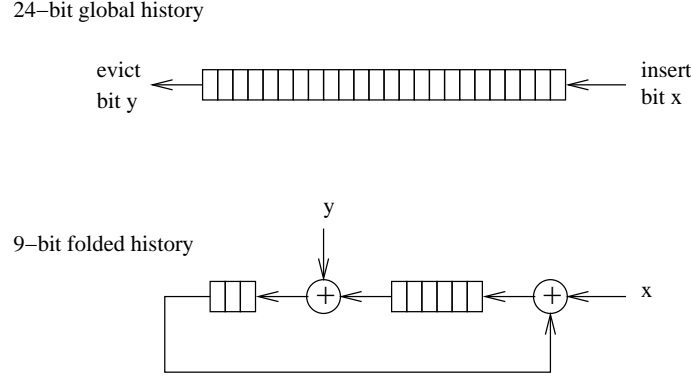
MATCH–SELECT MAJORITY–VOTE META–SELECT

Figure 5: Dealiasing primitives

24–bit global history

evict   insert
bit y   bit x

9–bit folded history

y

x

Figure 6: Global history compression : example of a 24-bit global history folded onto 9 bits.

a method for updating the input predictors. In the following, dealiased predictors will be described by specifying two-bit counter tables and using the primitives depicted on Figure 5.

Two-bit counter tables are defined either by describing a hashing function, or as the result of a primitive on a two-bit counter table. Two-bit counter table predictors often used are :

$$bimodal[A, m] = A \mod 2^m$$
$$gshare[A, H_h, m] = \left(A \oplus (H \times 2^{m-h}) \right) \mod 2^m$$

with A the branch address, H_h the global history of length h, and $\oplus$ the bit-wise XOR operation.

Global history hashing. When $h > m$, it is necessary to compress the global history. In this study, we compressed the global history in the following way :

$$fold(H_h, m) = \bigoplus_{j \geq 0} H_h \times 2^{-jm} \mod 2^m$$

It should be noted that global history folding is simple to implement. Updating the compressed history involves only two input bits : the bit inserted in the global history and the bit evicted from it. Figure 6 shows an example of a 24-bit global history folded onto 9 bits. This implementation requires to maintain and checkpoint both the non-compressed and compressed versions of the global history.

Note that when $h > m$, global history aliasing may occur, i.e., two distinct global history values may result in the same compressed value. Whatever hashing function, we cannot avoid

aliasing. A good hashing function must be simple to implement and distribute global history values seen by a static branch evenly onto table entries

An interesting property of history folding is that, when considering two distinct global history values, if these values only differ at bit positions located within a group of m consecutive bits, compressed values are distinct (path locality). A possible weakness of history folding concerns regular patterns with period equal to m. Another possible weakness concerns history values generated by bit rotations on a given value. This may happen for example on loop branches. To overcome this problem, it may be useful to add a level of hashing, like for example

$$hash\_fold(H_h, m) = fold(H_h, m) \oplus \Big(2 \times fold(H_h, m - 1) \Big)$$

Unless otherwise specified, we used a simple $fold(H_h, m)$ in our simulations.

The bias-split primitive. The *bias-split* primitive is extracted from the bi-mode predictor [22] (cf. Figure 5). It takes as input a two-bit counter table T and a bias predictor, and it returns T :

$$bias\text{-}split(bias\_predictor, table)$$

The bias-split primitive duplicates the two-bit counter table given as input. Both copies use the same hashing function, hence, physically, this is implemented with a single table (the multiplexor depicted on Figure 5 can be viewed as part of the addressing logic).

The bias prediction is used to select which half of the table will be used for the final prediction. The two-bit counter selected for the overall prediction is always updated with the branch outcome. If the overall prediction is correct but the bias prediction disagrees with the branch outcome, the bias predictor is not updated. Otherwise, the bias predictor is updated with the branch outcome. Hence, assuming the bias predictor implements a partial update (which is the case if the bias predictor is a single-table predictor), the overall predictor implements a partial update.

The bi-mode predictor [22] can be described as

$$
\boxed{
\begin{aligned}
&T_1 = bimodal[A, m] \\
&T_2 = gshare[A, H_h, m - 1] \\
&bi\text{-}mode[h, m] = bias\text{-}split(T_1, T_2) \\
&\textbf{size: } 2^m \times 4 \text{ bits}
\end{aligned}
}
$$

The bias-agree primitive The *bias-agree* primitive was introduced in [35]. It takes as input a bias predictor and a two-bit counter table T, and it returns T :

$$bias\text{-}agree(bias\_predictor, table)$$

The bias-agree primitive simply changes the way information is coded. Instead of predicting directly the outcome of the branch, the two-bit counter table predicts whether or not the

branch outcome will agree with the bias prediction. This can be done, for example, by an XOR with the bias prediction.

   The two-bit counter is updated accordingly : if the prediction is correct, it is strengthened, else it is weakened.

   If the overall prediction is correct but the bias prediction disagrees with the branch outcome, the bias predictor is not updated. Otherwise, the bias predictor is updated with the branch outcome. Hence, assuming the bias predictor implements a partial update, the overall predictor also implements a partial update.

   One of the simplest predictor based on this primitive is *agree-pu* (*pu* means *partial update*)

$$
\begin{array}{|l|}
\hline
T_1 = bimodal[A, m] \\
T_2 = gshare[A, H_h, m] \\
agree\text{-}pu[h, m] = bias\text{-}agree(T_1, T_2) \\
\textbf{size:}\ 2^m \times 4\ \text{bits} \\
\hline
\end{array}
$$

The majority-vote primitive.   The majority-vote primitive takes as input three predictors :

$$majority\text{-}vote(predictor1, predictor2, predictor3)$$

The overall prediction is obtained by a majority vote on the three predictions. Upon a misprediction, the three predictors are updated with the branch outcome. The majority-vote primitive implements a partial update : upon a correct prediction, we only update predictors which gave a correct prediction, i.e., if two predictors are correct but the third is wrong, the third predictor is not updated.

   The majority-vote primitive was first used in the *gskew* family of predictors [24, 31], which perform a majority vote on three two-bit counter tables. In practice, for these predictors to be efficient, the tables indexed with the same global history length must use different hashing functions. We define 3 extra hashing functions *gshare1*, *gshare2* and *gshare3*. For a fixed table size, it is not difficult to find three good hashing functions that will not increase significantly the predictor access time (for example, one can use the functions defined in [24]). For the purpose of simulation, wishing to obtain good results on a wide range of predictor table sizes and global history lengths, we defined the following three hashing functions :

$$
\begin{aligned}
gshare1[A, H_h, m] &= gshare[A \oplus (A/2^4), fold(H_h, m), m] \\
gshare2[A, H_h, m] &= gshare[A \oplus (A/2^8), fold(H_h, m-1), m] \\
gshare3[A, H_h, m] &= gshare[A \oplus (A/2^{12}), fold(H_h, m-2), m]
\end{aligned}
$$

From our experimentations, these functions have a good average behavior. The e-gskew predictor [24] can be described as follows :

$$T_1 = bimodal[A, m]$$
$$T_2 = gshare1[A, H_h, m]$$
$$T_3 = gshare2[A, H_h, m]$$
$$e\text{-}gskew[h, m] = majority\text{-}vote(T_1, T_2, T_3)$$
size: $2^m \times 6$ bits

The meta-select primitive. The meta-select primitive was introduced in [23]. It takes as input two predictors and a two-bit counter table :

$$meta\text{-}select(predictor1, predictor2, meta\_table)$$

The two-bit counter table delivers a *meta-prediction*. The meta-prediction is used to select which of the two predictors will give the overall prediction.

Unless stated otherwise, the meta-select we use in the remaining implements a partial update : **upon a correct prediction, we update only the predictor which was selected**.

As in [23], the meta-predictor is updated only when the two predictors give different predictions : in this case, if the prediction was correct, the two-bit selector is strengthened, else it is weakened.

As in [23], we define a *bimodal/gshare* hybrid in the following way :

$$T_1 = bimodal[A, m]$$
$$T_2 = gshare[A, H_h, m]$$
$$T_{meta} = gshare3[A, H_{h'}, m]$$
$$bimodal\text{-}gshare[h, h', m] = meta\text{-}select(T_1, T_2, T_{meta})$$
size: $2^m \times 6$ bits

One should note that this predictor **is not** the usual bimodal/gshare hybrid. First we use a partial update, whereas in [23] both predictors are always updated. Second, the hashing function used on the meta-predictor is different from those used on the two predictors (in [23] the meta-predictor uses the same index as the bimodal predictor; in the hybrid predictor of the Alpha 21264 [21], the meta-predictor uses the same index as the global history predictor).

Actually, for partial update to work well, it is necessary to use a different hashing function. For example let us consider two branches that are in conflict in the bimodal predictor. This conflict may be solved by predicting one branch with the bimodal predictor and the other branch with the gshare predictor. Thanks to partial update, the bimodal predictor entry will be dealiased. But for this to be possible, the two branches must be mapped on different meta predictor entries.

The 2bc-gskew predictor introduced in [31] also uses the meta-select primitive :

$$T_1 = bimodal[A, m]$$
$$T_2 = gshare1[A, H_h, m]$$
$$T_3 = gshare2[A, H_h, m]$$
$$T_{meta} = gshare3[A, H_h, m]$$
$$2bc\text{-}gskew[h, m] = meta\text{-}select(majority\text{-}vote(T_1, T_2, T_3), T_1, T_{meta})$$
size: $2^m \times 8$ bits

The match-select primitive. The *match-select* primitive is extracted from the YAGS predictor [7]. A quite similar primitive was used in the hybrid next-trace predictor proposed in [16] for the trace processor [29].

The match-select primitive takes as input a two-bit counter table and a default predictor :

$$match\text{-}select(table, default\_predictor)$$

This primitive augments each entry of the input two-bit counter table with a tag. This tag represents a few bits from the branch address. Upon a match with the branch address, the corresponding two-bit counter is used as the final prediction, else the default prediction is used. If there was a match, the corresponding two-bit counter is updated with the branch outcome else, if the default prediction is wrong, the missing tag is written in the table and the corresponding counter is initialized according to the branch outcome.

Note that match-select implements a partial update : no entry is allocated in the tagged table as long as the default prediction is correct (also, when the default prediction agrees with the branch outcome, it is updated with the branch outcome, else if it disagrees with the branch outcome, it is updated only if the overall prediction is wrong).

In practical implementations of the match-select primitive, tags are short. Theoretically, with t tag bits, the probability of aliasing on the tag value is 2^{-t}.

Unlike in [7], to prevent any problem, we will hash the address bits for obtaining the tags. Low-order address bits are often used as index bits (typically, gshare with a short global history) and a conflict on a table index may coincide with a conflict on the tag value.

The original YAGS predictor as described in [7] is not the simplest utilization of the match-select primitive. A simpler one, which we define as *gtags*, is

$$T_1 = gshare[A, H_h, m]$$
$$T_2 = bimodal[A, m]$$
$$tag = gshare3[A, 0, 6]$$
$$gtags6[h, m] = match\text{-}select(T_1, T_2)$$
size: $2^m \times 10$ bits

The "6" appended to *gtags* means that we use 6-bit tags. The YAGS predictor [7] can be described as

$$T_1 = gshare[A, H_h, m - 1]$$
$$T_2 = bimodal[A, m]$$
$$tag = gshare3[A, 0, 6]$$
$$YAGS6[h, m] = match\text{-}select(bias\text{-}split(T_2, T_1), T_2)$$
size: $2^m \times 10$ bits

6   Experimental Study of Dealiasing Primitives

In this section, we conduct an experimental study of the dealiasing primitives described in Section 5.2. Unless stated otherwise, two-bit counter table entries were initialized randomly, 50% in state 0 and 50% in state 3.

6.1   The Bias-Split, Bias-Agree and Bias-Inject Primitives

The bias-agree and bias-split primitives are very close in the way they are used and behave. The bias-split and bias-agree primitives tackle the aliasing problem with the same lever : turning destructive aliasing into harmless aliasing [35, 22].

The bi-mode and agree-pu predictors provide, in general, a better prediction accuracy than a single gshare table. This can be understood with Formula 7 : $m_{alias} \approx b(1 - b)n/N$. With $b = 0.5$, we have $m_{alias} \approx 0.25n/N$. If we double the size of a gshare table, we divide destructive aliasing by two. On the other hand, if instead of doubling the gshare table we implement a bias predictor so that $b = 0.1$, then $m_{alias} \approx 0.09n/N$. We divide destructive aliasing by almost three.

In this section, we compare the bias-split and bias-agree primitives. We show experimentally that bias-split is slightly more efficient than bias-agree. Figures 18 and 19 show the misprediction percentage for a bi-mode and an agree-pu predictor having a fixed gshare table (4k entries, $h = 10$) and a bimodal table which size is varied from 256 to 32k entries The gshare table was initialized optimistically both for bias-split and bias-agree.

We observe that aliasing in the bias predictor impairs significantly the overall prediction accuracy for both primitives. However, agree-pu is more sensitive to aliasing in the bias predictor than bi-mode (Figure 18, small bimodal predictor size). Also, it can be observed that, even when there is little aliasing in the bimodal table, bi-mode keeps an advantage over agree-pu (*mcf, twolf*).

Analysis.   Sometimes, the bias predictor mispredicts the **bias** of a branch. Bias misprediction happens mainly upon destructive aliasing in the bias predictor and for branches that are weakly biased. In general, a bias misprediction is a branch misprediction (note that branch mispredictions occur more often than bias mispredictions, $m_\infty > P_\infty$ when $p < 0.5$, cf. Section 2.1). However, this does not mean that the branch is unpredictable. It may be predicted correctly using a different hashing function or a longer global history.

On the bi-mode predictor, upon a bias misprediction, the gshare predictor is able to give a correct prediction, at the cost of using two table entries instead of one. On the other hand, the agree-pu predictor is able to correct a bias misprediction only if the occurrence of a global history value always coincides with a fixed bias prediction (either correct or wrong).

To show that this analysis is correct, we have performed two experiments. Figures 20 and 21 show the results obtained when two-bit counters in the bimodal predictor are replaced with three-bit counters. The use of three-bit counters in the bimodal predictor decreases the number of bias mispredictions. Now bi-mode and agree-pu have very close behaviors. Said differently, the use of three-bit counters in the bias predictor benefits more to bias-agree than to bias-split.

Figures 22 and 23 show the results obtained when we use a $gshare[A, H_{10}, m]$ table for the bias predictor instead of a bimodal table, with m varied from 13 to 19. Now the behaviors of bias-split and bias-agree are very close because the occurrence of a global history value often coincides with a fixed bias prediction.

A new primitive : bias-inject.   A possible weakness of bias-split is that, if branch biases are not distributed evenly between *taken* and *not-taken*, one half of the split table will be less used than the other half. This is not a problem with bias-agree. This weakness of bias-split does not appear on our benchmarks because the bias distribution is nearly balanced. However, an optimizing compiler may organize the program control flow so that branches are more often not-taken than taken [2]. For codes produced by such compiler, the behavior of bias-split may be degraded

It is possible to combine the strength of bias-split and bias-agree in a new primitive : *bias-inject*. This primitive is depicted on Figure 5. It takes as input a bias predictor and a two-bit counter table T, and it returns T :

$$bias\text{-}inject(bias\_predictor, table)$$

Like bias-agree, bias-inject records in the table whether or not the branch direction agrees with the bias prediction. It uses a partial update, like bias-agree and all the primitives introduced previously. The difference with bias-agree is that the bias prediction is concatenated with the global history so that it is hashed like a global history bit. We define the *agree-bimode* predictor :

$$
\begin{array}{l}
T_1 = bimodal[A, m] \\
T_2 = gshare[A, H_h, m] \\
agree\text{-}bimode[h, m] = bias\text{-}inject(T_1, T_2) \\
\textbf{size: } 2^m \times 4 \text{ bits}
\end{array}
$$

Like bias-agree and bias-split, bias-inject turns aliasing harmless in the gshare table. Like bias-split, it is able to correct a bias misprediction even if the occurrence of a global history value does not always coincides with the same bias prediction. But unlike bias-split, bias-inject makes no assumption on the bias distribution.

Figures 24 and 25 compares bi-mode and agree-bimode predictors. As can be observed, bias-split and bias-inject have very similar behaviors.

Conclusion. We have shown that the bias-split and bias-agree primitives are very similar, with bias-split being slightly more efficient than bias-agree. We introduced a new primitive, bias-inject, which combines the strength of bias-split and bias-agree. These primitives allow to build simple 2-table dealiased predictors like bi-mode.

6.2 The Meta-Select and Majority-Vote Primitives

The meta-select and majority-vote primitives are methods for choosing between two predictors P1 and P2. Generally, predictor P2 uses a global history shorter than P1. We want to determine which of P1 or P2 is likely to give the more accurate prediction for a particular branch. Meta-select uses a two-bit meta predictor for choosing between P1 and P2, whereas majority-vote uses a third predictor P3.

The differences between meta-select and majority-vote are subtle. We try in this section to characterize them.

Experiment 1. Figures 26 and 27 show the result of an experiment emphasizing one of the difference between meta-select and majority-vote. Here we are simulating three 4k-entry tables

$$T_1 = bimodal[A, 12]$$
$$T_2 = gshare1[A, H_h, 12]$$
$$T_3 = gshare2[A, H_h, 12]$$

And we compare the following predictors

$$meta\text{-}select(T_1, T_2, T_3)$$
$$majority\text{-}vote(T_1, T_2, T_3)$$
$$majority\text{-}vote(T_1, T_2, bias\text{-}agree(T_1, T_3))$$

As can be observed, when the global history is small, meta-select and majority-vote behave similarly. However, as the global history is increased beyond 10 bits, the misprediction percentage is degraded faster with majority-vote than with meta-select. With meta-select, as most branches are predicted correctly with the bimodal table T_1, aliasing in the meta-predictor T_3 is mostly harmless. On the other hand, with majority-vote, aliasing in table T_3 is 50% destructive. By applying bias-agree on table T_3, majority-vote becomes almost as good as meta-select. The equivalence can be easily verified with boolean identities :

$$p_{meta} = \overline{p_3}p_1 + p_3p_2$$

Now let us consider a majority-vote

$$p_{vote} = p_1p_2 + p_1p_3' + p_2p_3'$$

where p_3' is obtained by a bias-agree of p_1 over p_3 (let us assume bias-agree is implemented with an exclusive-or)

$$p_3' = p_1 \oplus p_3$$

If $p_3 = 1$, $p_3' = \overline{p_1}$ and $p_{meta} = p_2 = p_{vote}$. If $p_3 = 0$, $p_3' = p_1$ and $p_{meta} = p1 = p_{vote}$. This shows that prediction information is coded exactly in the same way.

After applying bias-agree on table T_3, majority-vote and meta-select differ only in the way tables are updated. Meta-select updates the tables less often than majority-vote. This concerns tables T_1 and T_2 (on a correct prediction, majority-vote reinforces both predictors if they both gave a correct prediction, whereas meta-select reinforces only the predictor which was selected) as well as table T_3 (when T_1 and T_2 give identical predictions, meta-select does not update T_3). This "minimal" partial update in meta-select helps decrease aliasing effects.

Experiment 2.   Figures 28 and 29 show another experiment highlighting the differences between meta-select and majority-vote. Here, the global history length is kept fixed to 10 on table T_2, and it is varied from 0 to 10 on table T_3.

The interesting point is for $h = 0$ on table T_3. As expected, majority-vote has a bad behavior. On the other hand, meta-select is able to take advantage of the 10-bit global history on T_2.

When we apply bias-agree on table T_3, the behavior of majority-vote is greatly improved. It should be noted that this effect does not come from a reduction of destructive aliasing : with $h = 0$, there is little aliasing on table T_3.

Let us consider a branch which is perfectly predicted by gshare and **often** mispredicted by the bimodal predictor. On a simple majority-vote, we would like table T_3 to give a prediction different from table T_1, so that the overall prediction is decided by the gshare table T_2. Such state is a stable state thanks to partial update. However, with $h = 0$ on table T_3, majority-vote alone is not able to reach such stable state because T_1 and T_3 always deliver identical predictions. By utilizing a primitive like bias-agree [5] on table T_3, we skew the updating of T_1 and T_3. For branches which are often mispredicted by the bimodal table, there is statistically a good chance to succeed in obtaining different predictions from T_1 and T_3 and put the predictor in a stable state.

As can be observed on Figure 28, even improved with a bias-agree, majority-vote is still slightly less efficient than meta-select. Let us consider a branch which is perfectly predicted by gshare and **sometimes** mispredicted by the bimodal table. For reversing the prediction of T_1 or T_3, we need consecutive mispredictions (partial update, on majority-vote, reinforces a predictor when it gives a correct prediction). For some branches, this may never happen. In this case, majority-vote cannot reach the stable state. On the other hand, meta-select does not update T_3 when T_1 and T_2 give identical predictions, in particular correct predictions : on each misprediction, the predictor gets closer to the stable state.

---

[5]we observed a similar effect with bias-split

Now let us analyze the impact of the global history length on table T_3. As expected, on majority-vote, increasing the global history on table T_3 increases the chance to reach a stable state. However, it also improves the behavior of meta-select (cf. *video_play* on Figure 29). The following experiment allows to better understand this observation.

Experiment 3. In this experiment, we study the behavior of $meta\text{-}select(T_1, T_2, T_3)$, with $T_1 = bimodal[A, m]$, $T_2 = gshare1[A, H_{10}, m]$ and $T_3 = gshare2[A, H_h, m]$, for $m = 12$ (3x4k) and $m = 16$ (3x64k).

Figures 30 and 31 show the misprediction percentage when h, i.e., the global history length on the meta-predictor, is varied from 0 to 10. The meta-select primitive described in this study uses a partial update, whereas the original meta-select [23] uses a total update (both predictors are always updated). On Figure 30, we show results for both partial update ("pu") and total update ("tu").

When aliasing is significant (3x4k), partial update is better than total update because it decreases aliasing. However, when there is little aliasing (3x64k), total update is better, especially for small h values (*video_play*).

Let us consider a branch which is perfectly predictable by gshare. Let us assume the branch is currently predicted by bimodal (for example 90% correctly), and bimodal mispredicts the branch for a certain global history value. With $h = 0$, it is theoretically possible to predict the branch 100% correctly by changing the meta-predictor state so as to select gshare instead of bimodal for that static branch.

However, with partial update, gshare will be updated only when the bimodal predictor mispredicts the branch. This means that gshare will be updated only for the global history value corresponding to bimodal predictor failures, but will not be updated for other global history values. Hence, with $h = 0$, the meta-predictor is not able to select gshare for that static branch because bimodal is correct more often than gshare. On the other hand, with total update, the meta-predictor is able to change its selection.

With partial update, this problem can be partly solved by using the same global history length on the meta-predictor and on the gshare table (cf. *video_play* on Figure 31). In this case, the meta-predictor selects gshare only for the global history value corresponding to bimodal predictor failures.

More generally, as aliasing in the meta-predictor is mostly harmless, increasing the global history length on the meta-predictor is often beneficial. A given branch may be better predicted by gshare for certain global history values, and better predicted by bimodal for others, either because of aliasing in the gshare table or because of periodic branch behaviors.

The 2bc-gskew predictor. The 2bc-gskew predictor (cf. Section 5.2), is a 4-table predictor combining the meta-select and majority-vote primitives. On Figures 32 and 33, we compare a 4-KByte 2bc-gskew ("meta/vote") predictor with a simple 4-KByte hybrid predictor ("meta") . More precisely, the 2bc-gskew predictor is

$$meta\text{-}select(majority\text{-}vote(T_1, T_2, T_3), T_1, T_{meta})$$

with

$$T_1 = bimodal[A, 12]$$
$$T_2 = gshare1[A, H_h, 12]$$
$$T_3 = gshare2[A, H_h, 12]$$
$$T_{meta} = gshare3[A, H_h, 12]$$

The hybrid predictor is

$$meta\text{-}select(T_1, T_2', T_3)$$

with

$$T_2' = gshare1[A, H_h, 13]$$

We observe that a 4-table 2bc-gskew is slightly better than a simple 3-table hybrid. This slight improvement comes from a better identification of (B, H_h) pairs with a long reuse distance, these branches being then predicted with the bimodal table.

To understand this, let us consider a gshare table, and a branch with a long reuse distance. This branch has a high probability of being mapped on an aliased entry of the gshare table. Let us assume this probability is close to 1. The probability to get a wrong prediction from the gshare table is roughly 1/2 (destructive aliasing). Now, we split the gshare table in two half-smaller tables, and we decide to use the gshare prediction only when the two tables give identical predictions (this is what is done in 2bc-gskew). The probability of aliasing for a branch with a long reuse distance is close to 1 on both tables. But the probability to get a wrong prediction from the gshare tables is only 1/4. As can be seen on Figures 32 and 33, this is useful when the global history is long.

Note that it is possible to replace the majority-vote in 2bc-gskew by a meta-select ("meta/meta") :

$$meta\text{-}select(meta\text{-}select(T_1, T_2, T_3), T_1, T_{meta})$$

As can be observed on Figures 34 and 35, "meta/vote" is practically equivalent to "meta/meta". The first meta-predictor T_{meta} detects branches that really need to be predicted with gshare. However, because of aliasing in T_{meta}, certain branches that would be better predicted by the less-aliased bimodal table are inopportunely steered to gshare. The second meta-predictor T_3 acts as a corrector for the first one, by putting back into the bimodal table branches which were inopportunely evicted from it by the first meta-predictor.

Conclusion. Meta-select seems a more general primitive than majority-vote. We were able to find configurations (3-table predictors) where replacing majority-vote by meta-select is clearly advantageous, but no configurations where majority-vote is clearly advantageous.

6.3 The Match-Select Primitive

The match-select primitive performs roughly the same work as meta-select, but in a very different way.

Figures 36 and 37 show the misprediction percentage of $YAGS6[h, 12]$ and $gtags6[h, 12]$ for a global history length h varying from 0 to 64 bits. It can be observed that these two predictors, both based on the match-select primitive, have very similar behaviors, *gtags* being marginally better than $YAGS$.

In fact, there is no clear justification for combining match-select with bias-split, as in YAGS. The purpose of bias-split is to make aliasing harmless. However, as a 2-bit counter in the tagged table is owned by a single branch at a time (provided tags are long enough), bias-split is practically useless.

Global history aliasing. As noted previously in Section 5.2, when the global history length is greater than the number of indexing bits, global history aliasing may occur. Examples of global history aliasing appear on Figure 37 on *video_play* for $h = 50$ and on *verilog* for $h > 44$. For *video_play*, it is possible to remove this particular instance of global history aliasing by changing the hashing function, for example using *gshare2* instead of *gshare* (note that this instance of aliasing concerns $gtags[50, 12]$, but we also observed it on $YAGS[50, 13]$).

The instance of global history aliasing observed on *verilog* concerns a loop branch which "sees" global history values generated by bit rotations. As noted previously in Section 5.2, this type of aliasing can be removed by replacing $fold(H_h, m)$ with $hash\_fold(H_h, m)$. In all subsequent simulations, we used $hash\_fold$ [6].

Match-select vs. meta-select. To better understand what distinguishes match-select and meta-select, we compare a gtags4 predictor

$$T_1 = bimodal[A, m]$$
$$T_2 = gshare[A, H_h, m]$$
$$tag = gshare3[A, 0, 4]$$
$$gtags4[h, m] = \text{match-select}(T_2, T_1)$$

with a meta/meta predictor

$$T_1 = bimodal[A, m]$$
$$T_2 = gshare1[A, H_h, m]$$
$$T_{meta1} = gshare2[A, H_h, m]$$
$$T_{meta2} = gshare3[A, H_h, m]$$
$$\text{meta-select}(\text{meta-select}(T_1, T_2, T_{meta1}), T_1, T_{meta2})$$

Note that these two predictors are comparable. First they have equal size. Second, they feature the same quantity of "bimodal" prediction, "gshare" prediction, and "selection"

---

[6] we did not observe a significant impact on benchmarks other than *verilog*

information. For example, with a total 4-KByte budget, meta/meta and gtags4 both feature 1 KB of "bimodal" prediction, 1 KB of "gshare" prediction, and 2 KB of "selection" information.

The result of this comparison is presented on Figures 38,39, 40 and 41, for a total budget of 4 KBytes ($m = 12$) and 64 KBytes ($m = 16$). We also show the misprediction percentage of a gtags6 (i.e., 6-bit tags), for a budget of 5 KBytes and 80 KBytes.

On Figures 40 and 41, we show two versions of meta/meta : one initializes the meta-predictors randomly ("rand", i.e., the default initialization), the other initializes table T_{meta2} so that T_1 is selected and table T_{meta1} so that T_2 is selected ("bim/gsh").

Match-select and meta-select differ in the way they behave under aliasing. The advantage of match-select appears on cold-start and capacity aliasing.

- **Cold-start aliasing.** On the first occurrence of a global history value, match-select naturally selects the bimodal table, which delivers a reasonably accurate prediction. Meta-select, on the other hand, may incur a significant fraction of cold-start mispredictions ("bim/gsh" vs. "rand"). However, it should be noted that the "rand" version of meta/meta is overly pessimistic : unless the program previously running on the machine was badly predicted with the bimodal table, table T_{meta2} should be in a state in which the bimodal table is selected for a majority of branches.

- **Conflict aliasing.** Conflict aliasing corresponds to a situation such that only a small fraction of static (B, H_h) pairs are mapped on 2-load entries. This corresponds to a small x in Formula 6. In this situation, a meta-predictor is able to identify aliased entries and discard them, keeping the benefit of a majority of entries that are not aliased. Moreover, being adaptive, the meta-predictor is able to take advantage of harmless aliasing. It can be observed on Figure 38 that when the predictor does not suffer too much from cold-start and capacity aliasing, a double meta-predictor ("meta/meta") may be more efficient than a 4-bit tag (*nroff, sdet, video_play*).

- **Capacity aliasing.** Capacity aliasing corresponds to a situation where a significant fraction of static branches are aliased. Temporal locality is crucial in this situation. An entry may be aliased at some times, and not aliased at other times. Referring to the multi-set model suggested at the end of Section 5.1, a branch may be aliased in set S_2 and not aliased in set $S_1 \subset S_2$. Capacity aliasing occurs when set S_2 is large and contribute to a significant fraction of dynamic branches. As long as the x in Formula 6 is greater than 4, doubling the table size, i.e., dividing x by 2, hardly decreases the number of aliased branches. In this situation, match-select is very efficient because it is able to detect when an entry can be used safely. On the other hand, meta-select is not very efficient because it does not know when an entry should be used or not. This weakness of meta-select appears on Figures 38,39, 40 and 41 when the global history is long. It also appears very clearly for *real_gcc*.

  To corroborate the analysis, Figures 42 and 43 show the percentage of dynamic branches which use the gshare predictor on a 4-KByte meta/meta and a 5-KByte

gtags6. The figure also shows the percentage of T_{meta1} updates in meta/meta that result in a change of the meta-prediction. First, we observe that match-select often permits using the gshare predictor more often than meta-select. For example, with a 10-bit global history, almost 50% of predictions are obtained from the gshare predictor when match-select is used, which is roughly twice higher than when meta-select is used. As can be observed, for *real_gcc*, *go* and *twolf*, table T_{meta1} experiences a significant fraction of meta-prediction transitions. This is a symptom of the meta-predictor hesitating between bimodal and gshare, gshare being more accurate than bimodal at certain times (non aliased entry), and less accurate at other times (aliased entry).

Conclusion. The match-select primitive is characterized by its ability to obtain accurate predictions out of aliased entries. Practically, match-select can only improve the prediction accuracy : in the worst case, all accesses to the tagged table are misses, and predictions are given by the default predictor. An other advantage of match-select is that it is very effective after context switches [31].

6.4   Approximating BPPM by Cascading Several Match-Select

From Figures 40 and 41, even with a large hardware budget, it is difficult to exploit a global history greater than 30 with a gtags or a 2bc-gskew, especially for benchmarks with a large working set like *real_gcc*, *go* and *twolf*. From our experiments, even with a larger hardware budget and wider tags, gtags is limited.

One limitation is the huge number of misses for very long global history lengths. These are cold-start misses (cf. Figures 12 and 13). and capacity misses. For example, let us assume 10% of dynamic branches miss the gshare table. If the default prediction is obtained from a bimodal table having a 90% prediction accuracy, then 1% of dynamic branches are mispredictions on misses. If the misprediction ratio is already very low, for example 2%, misses contribute to 50% of mispredictions.

This problem can be alleviated by using, as default predictor, a predictor more accurate than a bimodal table, for example a gtags using a moderately long global history. One could view such predictor as a "second order" approximation of BPPM.

On Figures 44 and 45, we compare a BPPM predictor and a predictor obtained by cascading two match-select :

$$T_1 = gshare1[A, H_h, m]$$
$$T_2 = gshare1[A, H_{h/4}, m]$$
$$T_3 = gshare1[A, 0, m]$$
$$gtags[h, h/4, m] = match\text{-}select(T_1, match\text{-}select(T_2, T_3))$$

The "middle" table T_2 uses a global history length which is one fourth the global history length h used on table T_1. We simulated three hardware budgets : 208 KBytes ($m = 16$, 10-bit tags), 20 KBytes ($m = 13$, 7-bit tags) and 9 KBytes ($m = 12$, 6-bit tag). With a 9 KB budget, it seems difficult to exploit global histories longer than 20. With a 20 KB budget,

we can exploit global histories up to 40 bits on the IBS. As we increase the hardware budget, we are able to exploit even longer global histories and get closer to a BPPM predictor.

Figure 46 compares BPPM and cascaded match-select when the maximum global history length is fixed to $h = 64$. On the x-axis, we vary the log_2 of the number of entries per table. The tag width is fixed to 10 bits. Three configurations are displayed : a 3-table configuration using global history lengths $(64, 16, 0)$, a 4-table $(64, 16, 8, 0)$ and a 5-table $(64, 32, 16, 8, 0)$.

Figure 47 shows the same comparison for the SPEC benchmarks with a maximum global history $h = 32$. The configurations simulated are a 3-table $(32, 8, 0)$, a 4-table $(32, 8, 4, 0)$ and a 5-table $(32, 16, 8, 4, 0)$

As can be observed, the gap between BPPM and cascaded match-select diminishes as we increase the number of tables and the number of entries per table. However, it seems difficult to close this gap with reasonable hardware budgets. Referring to the multi-set model suggested at the end of Section 5.1, the program can be modeled with several branch (more precisely "static" (B, H_h) pairs) sets $(S_i)_{i=1..n}$ contributing to a fraction d_i of dynamic branches. In general, the overall set S_n is very large, requiring a huge table. Although contribution d_n is usually negligible compared with 1, it may not be negligible compared with the misprediction ratio.

Remark. In principle, if we use a sufficient number of tables, we can get as close to BPPM as wished. However, in our experimentations, we observed that, as we increase the number of tables, cascaded match-select suffers from partial update. By running simultaneously partial update and total update on 65 tables, we observed a pathological behavior due to partial update.

For example, consider a branch which is strongly biased, but sometimes takes the direction opposite to the bias. As long as the branch is correctly predicted with $h = 0$, no entries are allocated in the other tables for this branch. When the branch takes the "bad" direction, it is mispredicted : entries are allocated in other tables, the corresponding two-bit counters being initialized with the "bad" outcome. This corresponds to a certain global history value H_{64}. If the next match is on H_{64}, a single misprediction is generated, and all counters with a "bad" prediction are corrected at once. However, if the next (longest) match is on level H_i, we correct only levels $j \leq i$, leaving the possibility of extra mispredictions generated by subsequent matches on levels $j > i$. When using few tables (for instance 3), partial update is not really penalizing, even for large budgets. When using many tables, total update is asymptotically better but requires very large hardware budgets.

6.5   Conclusion

We studied in this section six dealiasing primitives extracted from GHBPs previously proposed in the literature. Three of these primitives, bias-agree [35], bias-split [22], and bias-inject, are nearly equivalent. These three primitives allow to build simple dealiased predictors using only 2 tables, like the bi-mode predictor [22]. Such 2-table predictors are worth considering as L1 predictors.

More accurate (and more complex) dealiased predictors can be built using the selection primitives, majority-vote, meta-select and match-select. We have studied the differences between majority-vote [24] and meta-select [23], and we have shown that meta-select is generally a slightly more efficient selection primitive than majority-vote, because there is a bias-agree hidden in meta-select. Like for majority-vote, the dealiasing power of meta-select relies on the use of different hashing functions and a partial update [24].

The match-select primitive, extracted from the YAGS predictor [7], performs a selection like majority-vote and meta-select. Match-select is particularly effective at overcoming cold-start and capacity misses. We have shown that cascading two match-select allows to use very long global histories, such predictor being an approximation of BPPM. Practically, with such 3-table predictor, we overcome the history length dilemma [20, 36].

Directions for future studies. This section was focused more on studying predictor primitives than finding the "best" predictor. Existing publications, like the one previously cited, already provide a lot of experimental data.

Finding the more cost-effective way, given hardware constraints, to combine these primitives requires further research. One could consider, for example, a systematic approach like the one proposed in [8]. The set of dealiasing primitives we isolated is by no mean complete. It may be interesting to study 2-table and 3-table predictors in an systematic way, like was done for two-bit automatons in [26].

All the predictors studied in this section are approximations of BPPM, but the BPPM predictor studied in Section 4 can only react to the information it receives. In this study, we assumed the global history was a fixed input parameter, defined as the sequence of directions of dynamic branches encountered the most recently (as in most previous studies). But this is mainly for practical reasons. For example, we could use instead the global history of previous branch target addresses, as was proposed in [25, 16, 36].

Branch target addresses convey *a priori* more path information than branch directions. It would be interesting to determine, in the context of a BPPM predictor, whether or not a global history of target addresses really contains more information or is just equivalent to taking a longer global history of branch directions.

Also, it is generally assumed that the global history is continuous, i.e., represents consecutive branches. Indeed, all path information is not interesting. Actually, only certain branches convey interesting path information, i.e., information which helps predicting subsequent branches [9]. Other branches degrade the predictor behavior, increasing the working set unnecessarily and contributing to the discrepancy between the dashed and solid curves on Figures 16 and 17.

It might be possible, based on static analysis and profile information, to identify branches which do not contribute to interesting path information, so that these branches do not "pollute" the global history.

Trying to do this dynamically, based on general code properties, seems more difficult *a priori*, but may be interesting too. Experimentations on using a *return history stack* to restore, partially or totally, the global history after a function return [10, 16] seem to indicate

that it might be possible to introduce efficient discontinuities in the global history. It should be noted that, in the context of two-level predictors using per-branch local histories [41, 23], a local history can be viewed as a discontinuous global history.

7   Experimental Evaluation of the Perceptron Predictor

The idea to use neural network models for dynamic branch prediction is recent, mainly because their implementation in current commercial processors is not realistic.

It was argued in [18] that one of the simplest neural network model, the *perceptron*, may be considered for a feasible implementation. The authors studied a table of perceptrons used as a GHBP and showed that the perceptron is able to exploit very long global histories.

The perceptron works in a way very different from BPPM and predictors derived from it. This section provides new experimental results on the perceptron predictor.

7.1   Perceptron Description

The perceptron tries to learn the branch outcome for each global history value seen by the branch. In other words, it tries to determine for which global history values it should predict *taken*, and for which it should predict *not-taken*.

Information learned about the branch behavior is maintained in a set of $n + 1$ weights $(w_i)_{i=0..n}$. Each weight is a signed integer value, coded on a few bits. These $n + 1$ weights are recorded in the entry corresponding to the branch in a *perceptron table*. Ideally, there is an entry (i.e., a perceptron) for each static branch. However in practice, the number of table entries is limited, so several branches may share the same perceptron.

In the following description, the global history is coded as a sequence $(g_i)_{i=1..n}$ of branch outcomes being either 1 (taken) or -1 (not taken). For $i = 0$, we have

$$g_0 = 1$$

The following description is taken directly from [18].

Obtaining a prediction.   To obtain a prediction, we index the perceptron table with the branch address, and we get the set of $n + 1$ weights defining the perceptron state for that branch. From these weights (w_i) and the current global history (g_i), we compute the perceptron output as follows :

$$y = \sum_{i=0}^{n} g_i \times w_i$$

and the final prediction is given by the sign of y, i.e., we predict *taken* if $y \geq 0$ and *not-taken* if $y < 0$. Note that when $n = 0$, the prediction is given by the sign of the bias weight w_0.

Updating the weights. Once the branch outcome t is known ($t = -1$ or $t = 1$), the $n+1$ weights are updated as follows :

$$\text{if } sign(y) \neq t \text{ or } |y| \leq \theta \text{ then}$$

$$\forall i \in [0..n] \quad w_i \leftarrow w_i + t \times g_i$$

In other words, a weight w_i is incremented when the branch outcome t is equal to g_i, else it is decremented. Parameter θ is the *threshold*. The threshold decides when enough training has been done for a particular global history value.

Threshold value. In [18], the best threshold value θ was determined experimentally as a function of the number of weights

$$\theta = \lfloor 14 + 1.93n \rfloor \tag{9}$$

As in [18], weights are coded on $1 + \lceil log_2\theta \rceil$ bits. Notice that when $n = 0$, the perceptron table is equivalent to a bimodal predictor using 5-bit up-down saturating counters (there is a single weight w_0 per entry).

Initialization of weights. In all experimental results presented here, all perceptron weights were initialized to 0. When running the benchmarks without resetting the perceptron weights between benchmarks, we observed a slight degradation of the prediction accuracy for some benchmarks. However, it **is** possible to reset perceptrons before running an application, and we think initializing with zeros, as in [18], gives more general results.

7.2 Linear Separability and Perceptron Sharing

Each perceptron table entry holds $n + 1$ weights. If we use $h = 64$ global history bits as perceptron inputs, we have $n = 64$ and weights coded on 9 bits, for a total of 585 bits per entry. For example, a 64-entry table requires a 4.6 KBytes budget, not counting the logic for computing the perceptron output. Hence a realistic perceptron predictor will have only few entries, and many branches will have to share a perceptron with other branches. Like aliasing in classical branch predictors, perceptron sharing degrades the prediction accuracy.

A perceptron is able to learn a boolean function perfectly only if this function is *linearly separable*. We define the function f_p of a branch as the boolean function which maps each global history value in the working set of that branch onto the corresponding branch outcome, either *taken* or *not taken*.

Each global history value corresponds to a point in a n-dimensional space, which coordinates are given by the n global history bits ($+1$ or -1). We distinguish two sets : global history values H for which $f_p(H) = 1$ and those for which $f_p(H) = -1$. If these two sets can be separated, in the n-dimensional space, by a hyperplane, then the function f_p is linearly separable.
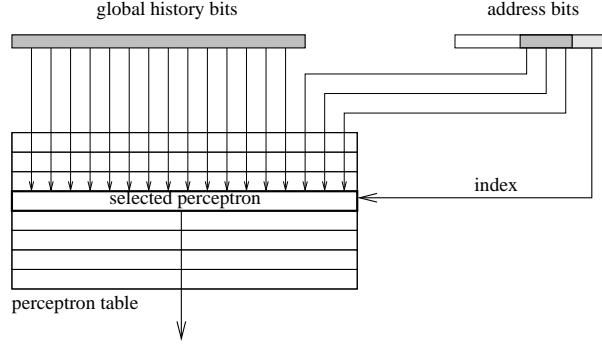
Figure 7: Using address bits as extra perceptron inputs

Branches that are strongly biased are mostly linearly separable. Problems arise for branches that are weakly biased and branches that share a perceptron. In the case of perceptron sharing, we can define a function f_p by merging the working sets of these two branches, assuming their working sets have a null intersection. If the two branches have the same bias, function f_p is roughly like that of a single branch with a working set twice larger. This is equivalent to harmless aliasing in classical predictors. On the other hand, if the two branches have opposite biases, the perceptron may fail to separate the two branch sets perfectly, thus generating mispredictions.

7.3   Injecting Address Bits

There are several ways to solve the perceptron sharing problem. We can increase the number of table entries. However, recalling the birthday paradox (cf. Section 5.1), this may not be the most cost-effective solution. A possible way to decrease perceptron sharing effects is to use a few address bits as extra inputs to the perceptron.

For instance, let us consider a perceptron table with 64 entries. This table is indexed with 6 low-order address bits. For some applications, there will be a lot of perceptron sharing. If the perceptron works on $h = 64$ bits of global history, each weight being coded on 9 bits, this requires a 4.6 KBytes budget. Now if we double the number of entries, we can hope to halve the number of conflicts, but this requires doubling the budget.

Another possibility is to concatenate some address bits with global history bits and use a slightly wider perceptron, as illustrated on Figure 7. In the remaining, we will denote a perceptron as

$$perceptron[m, H_h, A_a, ...]$$

where m is the log_2 of the number of table entries, H_h represents h global history bits, A_a represents a address bits, and the "..." represents any additional inputs.

On the previous example, if we concatenate $a = 8$ address bits to the global history bits, we have $n = h + a = 72$ inputs, and the budget increase is only $+12$ % (73/65).

Figures 48 and 49 show the misprediction interval of a perceptron predictor using a fixed number $h = 64$ of global history bits. We show four perceptron configurations

$$perceptron[m, H_{64}, A_a]$$

with a being the number of address bits used as inputs (these are address bits not used for indexing the table). Configuration $a = 0$ is the "normal" perceptron (65 weights per entry). Configuration $a = 12$ uses 12 address bits (77 weights per entry). The number of table entries is varied between 64 ($m = 6$) and 64k ($m = 16$). Perceptron weights are coded on 9 bits. For the IBS benchmarks, we also show the misprediction interval of BPPM with $h = 64$. For comparison with BPPM on the SPEC benchmarks, we show a similar experiment on Figure 50 using a global history length $h = 32$.

As can be observed, increasing the width of the perceptron is more cost-effective than increasing the number of table entries. Each address bit added in the inputs improves the linear separability. However, this solves the problem only partly.

Moreover, as was pointed out in [18], even when there is no perceptron sharing, the function f_p of certain branches (in particular branches that are weakly biased) is not linearly separable, which can explain the discrepancy between the perceptron and BPPM.

7.4   Injecting the Prediction from Another Predictor

It was proposed in [18] to solve the problem of linearly inseparable branches by combining a perceptron and a "classical" predictor, like gshare for example. In [18], a meta predictor was used to form a hybrid perceptron.

In this section, we use as "classical" predictor P_1 a bimodal/gshare using 10 bits of global history

$$
\begin{aligned}
T_1 &= bimodal[A, m] \\
T_2 &= gshare1[A, H_{10}, m] \\
T_3 &= gshare2[A, H_{10}, m] \\
P_1 &= meta\text{-}select(T_1, T_2, T_3)
\end{aligned}
$$

We will use the following hybrid perceptron :

$$
\begin{aligned}
T_4 &= gshare3[A, H_{10}, m] \\
P_2 &= perceptron[m - 6, H_{64}, A_8] \\
perceptron\text{-}hybrid &= meta\text{-}select(P_1, P_2, T_4)
\end{aligned}
$$

Note that the perceptron budget is roughly half the total budget, as in [18]. Figures 51 and 52 show the misprediction interval of the hybrid perceptron defined above, compared with that of a single $perceptron[m', H_{64}, A_8]$. Parameter m is varied from 12 to 18. On the graphs, we show two versions of the hybrid perceptron. One uses a partial update ("pu"), and the other a total update ("tu", P_1 and P_2 are always updated).
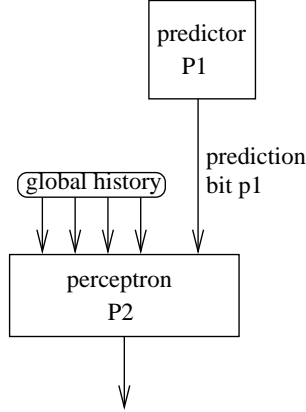
Figure 8: Example of a perceptron cascaded with a predictor P_1

Our results corroborate those of [18] : certain branches are better predicted with a classical predictor than with a perceptron. Figures 51 and 52 also show that partial update is better for "small" budgets and total update better for large budgets. Recall that a similar behavior was observed for a classical meta-select based predictor (cf. Figure 30).

Using a meta-select is not the sole way to combine the perceptron with another predictor P_1. Another way is to use the prediction p_1 from P_1 as an extra input to the perceptron, as illustrated on Figure 8. It should be noted that, in a processor using a perceptron as a L2 predictor, it is natural to use the L1 prediction for p_1. However, for comparisons, we will count both P_1 and P_2 in the hardware budget. Figures 53 and 54 compare the hybrid perceptron defined above with the following cascaded perceptron :

$$P_2' = perceptron[m - 6, H_{64}, A_8, p_1]$$

Both the hybrid and the cascaded perceptron use a total update, i.e., P_1 and P_2/P_2' are always updated. As can be observed, cascading is roughly equivalent to using a meta-predictor. Sometimes cascading is better (*verilog*), sometimes a meta-predictor is better (*twolf*). The advantage of cascading is that we do not need to implement a separate table for the meta-predictor.

Injecting prediction p_1 in the perceptron inputs may be analyzed from the point of view of linear separability. Originally, we have points of coordinates $(x_1, \cdots, x_n)$ in a n-dimensional space. Then we inject p_1, adding one dimension and one coordinate x_{n+1}. Hyperplane $x_{n+1} = 0$ separates the (n+1)-dimensional space in two half spaces. On one side of the hyperplane, we have (B, H) points predicted taken by P_1, and on the other side (B, H) points predicted not taken by P_1. This is, to some extent, analogous to the function performed by the bias-split primitive in a bi-mode predictor.

Generally, the cascaded perceptron P_2 is more accurate than the input predictor P_1, which corroborates the intuition that the perceptron is able to predict certain branches that classical predictors fail to predict correctly. Actually, the prediction accuracy of a cascaded perceptron depends on the prediction accuracy of P_1. If P_1 is the L1 predictor and the cascaded perceptron P_2 is the L2 predictor, then improving P_1 improves both prediction levels.

To illustrate this, we combine a perceptron with a 2-table gtags P_1 and a 3-table gtags P'_1 defined as

$$T_1 = gshare1[A, H_{64}, m]$$
$$T_2 = gshare1[A, H_{16}, m]$$
$$T_3 = gshare1[A, 0, m]$$
$$P_1 = gtags8[16, m] = match\text{-}select(T_2, T_3)$$
$$P'_1 = gtags8[64, 16, m] = match\text{-}select(T_1, P_1)$$

and we study the cascaded perceptrons

$$P_2 = perceptron[m - 6, H_{64}, A_8, p_1]$$
$$P'_2 = perceptron[m - 6, H_{64}, A_8, p'_1]$$

with p_1 and p'_1 the predictions from P_1 and P'_1 respectively. Both P_1/P'_1 and P_2/P'_2 are always updated (total update for the whole predictor), but predictor P_1/P'_1 uses a partial update. It should be noted that the size of $P_1 + P_2$ is approximately the size of P'_1. Hence we can compare directly the cascaded perceptron P_2 with the 3-table gtags P'_1, and evaluate the ability of the perceptron to improve over P_1.

Results are displayed on Figures 55 and 56. For allowing the comparison with BPPM on the SPEC benchmarks, we also simulated a global history $h = 32$, doubling the number of perceptron entries to keep approximately the same perceptron budget (Figure 57).

First, we observe that P'_1 is often better than P_2. In these cases, for a fixed hardware budget, match-select is better than the perceptron at improving over P_1 by looking at a longer global history. However for some benchmarks, especially *mpeg_play*, *mcf* and *twolf*, P_2 is better than P'_1.

Second, we observe that P'_2 is generally asymptotically better than P'_1 : given the *same* global history information, the perceptron is able to improve over a classical predictor.

It is interesting to note that, for some benchmarks, the perceptron helps bringing the prediction accuracy closer to that of BPPM. This is particularly striking on *mpeg_play*, which is, among the IBS benchmark, the most difficult to predict with classical predictors.

7.5   Research Directions

The perceptron offers a new way to tackle the branch prediction problem. Unlike most GHBPs proposed previously, the perceptron does not belong to the family of predictors derived from BPPM. On BPPM, if two global history values H and H' differ only at a single bit position, this is a miss, even if this bit corresponds to a branch bringing no

correlation information. On a perceptron, on the other hand, it is possible to predict H' with prediction information recorded for H.

The perceptron seems to be very sensitive to bit locality in the global history, that is, the fact that global history values differ only at certain bit positions. A strong hashing of global history values (i.e., a hashing which breaks bit locality), has no significant impact on a predictor derived from BPPM. On the perceptron, a strong hashing degrades the prediction accuracy significantly.

Further analysis is required to better understand the impact of bit locality on the perceptron ability to learn branch behaviors.

8   Conclusion

This study was a recapitulation of recent research on global history branch prediction. We emphasized the fact that most GHBPs previously proposed are approximations of BPPM. Dealiasing techniques, in this context, are necessary to bring storage capacities required to store branch prediction information down to reasonable sizes. We isolated six dealiasing primitives from GHBPs previously proposed and studied the characteristics of these primitives.

At last, we compared the perceptron, a technique adapted from neural networks, with predictors derived from BPPM. Our study, confirming that of [18], shows that the perceptron has a potential for improving the prediction accuracy given a fixed hardware budget.

Further studies are necessary to find other dealiasing primitives and new ways to combine these primitives. The use of the perceptron as an efficient branch predictor is still a research issue. This concerns both its feasibility in hardware and its ability to improve the prediction accuracy.

One aspect of global history prediction which is less understood (and which was not the concern of this study) is what information we should put in the global history, and how to represent this information. This concerns both the perceptron and predictors derived from BPPM.

References

[1] Doug Burger and Todd Austin. The Simplescalar toolset. Technical Report TR-97-1342, University of Wisconsin, Madison, June 1997.

[2] B. Calder and D. Grunwald. Reducing branch costs via branch alignment. In *Proceedings of the 6th International Conference on Architectural Support fro Programming Languages and Operating Systems*, 1994.

[3] B. Calder and D. Grunwald. Next cache line and set prediction. In *Proceedings of the 22nd International Symposium on Computer Architecture*, 1995.

[4] I.-C.K. Chen, J.T. Coffey, and T.N. Mudge. Analysis of branch prediction via data compression. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[5] J.G. Cleary and I.H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396–402, April 1984.

[6] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to algorithms*. MIT Press, 1990.

[7] A.N. Eden and T. Mudge. The YAGS branch prediction scheme. In *Proceedings of the 31rst Annual International Symposium on Microarchitecture*, 1998.

[8] J. Emer and N. Gloy. A language for describing predictors and its application to automatic synthesis. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[9] M. Evers, S.J. Patel, R.S. Chappell, and Y.N. Patt. An analysis of correlation and predictability: what makes two-level branch predictors work. In *Proceedings of the 25th International Symposium on Computer Architecture*, 1998.

[10] Marius Evers. *Improving branch prediction by understanding branch behavior*. PhD thesis, University of Michigan, 1999. CSE-TR-417-99.

[11] E. Federovsky, M. Feder, and S. Weiss. Branch prediction based on universal data compression algorithms. In *proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[12] W. Feller. *An introduction to probability theory and its applications*, volume 1. Wiley, second edition, 1957.

[13] R.L. Graham, D.E. Knuth, and O. Patashnik. *Concrete mathematics*. Addison-Wesley, 1989.

[14] E. Hao, P.-Y. Chang, and Y.N. Patt. The effect of speculatively updating branch history on branch prediction accuracy, revisited. In *Proceedings of the 27th International Symposium on Microarchitecture*, 1994.

[15] W.W. Hwu and Y.N. Patt. Checkpoint repair for out-of-order execution machines. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, 1987.

[16] Q. Jacobson, E. Rotenberg, and J.E. Smith. Path-based next trace prediction. In *Proceedings of the 30th International Symposium on Microarchitecture*, 1997.

[17] D.A. Jiménez, S.W. Keckler, and C. Lin. The impact of delay on the design of branch predictors. In *Proceedings of the 33rd International Symposium on Microarchitecture*, 2000.

[18] D.A. Jiménez and C. Lin. Dynamic branch prediction with perceptrons. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, January 2001.

[19] S. Jourdan, T.-H. Hsing, J. Stark, and Y.N. Patt. The effect of mispredicted-path execution on branch prediction structures. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 1996.

[20] 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 International Symposium on Computer Architecture*, 1998.

[21] R.E. Kessler. The Alpha 21264 microprocessor. *IEEE MICRO*, 19(2), March 1999.

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

[23] Scott McFarling. Combining branch predictors. Technical note TN-36, DEC WRL, June 1993.

[24] 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*, 1997.

[25] R. Nair. Dynamic path-based branch correlation. In *Proceedings of the 28th International Symposium on Microarchitecture*, 1995.

[26] R. Nair. Optimal 2-bit branch predictors. *IEEE Transactions on Computers*, 44(5):698–702, May 1995.

[27] S.T. Pan, K. So, and J.T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

[28] H. Patil and J. Emer. Combining static and dynamic branch prediction to reduce destructive aliasing. In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture*, 2000.

[29] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J.E. Smith. Trace processors. In *Proceedings of the 30th International Symposium on Microarchitecture*, 1997.

[30] S. Sechrest, C.-C. Lee, and T. Mudge. Correlation and aliasing in dynamic branch predictors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[31] André Seznec and Pierre Michaud. De-aliased hybrid branch predictors. Research report PI-1229, IRISA, France, February 1999.

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

[33] A. Sodani and G.S. Sohi. Understanding the difference between value prediction and instruction reuse. In *Proceedings of the 31st International Symposium on Microarchitecture*, 1998.

[34] http://www.spec.org.

[35] E. Sprangle, R.S. Chappell, M. Alsup, and Y.N. Patt. The agree predictor: A mechanism for reducing negative branch history interference. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[36] J. Stark, M. Evers, and Y.N. Patt. Variable length path branch prediction. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[37] A.R. Talcott, M. Nemirovsky, and R.C. Wood. The influence of branch prediction table interference on branch prediction scheme performance. In *Proceedings of the 3rd Annual International Conference on Parallel Architectures and Compilation Techniques*, 1995.

[38] R. Uhlig, D. Nagle, T. Mudge, S. Sechrest, and J. Emer. Coping with code bloat. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[39] T.-Y. Yeh and Y.N. Patt. Two-level adaptive branch prediction. In *Proceedings of the 24th International Symposium on Microarchitecture*, November 1991.

[40] T.-Y. Yeh and Y.N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, May 1992.

[41] Tse-Yu Yeh. *Two-level adaptive branch prediction and instruction fetch mechanisms for high performance superscalar processors*. PhD thesis, University of Michigan, 1993.

[42] C. Young, N. Gloy, and M.D. Smith. A comparative analysis of schemes for correlated branch prediction. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

A Theoretical Study of the Growth of a Working Set

Let S be a set of N elements. We consider a sequence of n elements chosen from S according to a certain probability function. We define the working set s_q of rank q as the subset of elements that have been chosen more than q times in the sequence. Its size is $w_q(n) \leq N$. In particular, $w_0(n)$ represents the number of unique elements in a sequence of length n.

A.1   Set of Equiprobable Elements

We assume all the elements in S have an equal probability to be chosen. Let us define $x = n/N$ and $v_q(x) = w_q(n)/N$.

First, we seek to evaluate $v_0(x)$. Let us increase the sequence by one element. There is a probability $v_0(x)$ that an element chosen randomly in S already belong to s_0. We have

$$w_0(n+1) - w_0(n) = 1 - v_0(x) = 1 - \frac{w_0(n)}{N}$$

which can be written

$$[w_0(n+1) - N] = [w_0(n) - N] \times \left(1 - \frac{1}{N}\right)$$

then

$$w_0(n) - N = [w_0(0) - N] \times \left(1 - \frac{1}{N}\right)^n$$

For $N > 1$, we have $(1 - 1/N)^n \simeq e^{-n/N}$, hence

$$w_0(n) \simeq N(1 - e^{-n/N})$$

$$v_0(x) \simeq 1 - e^{-x} \tag{10}$$

We can deduce v_q recursively for $q > 1$. Let us increase the sequence by one element. The probability to increase s_q is equal to the fraction of elements that have already been encountered exactly q times

$$w_q(n+1) - w_q(n) = v_{q-1}(x) - v_q(x)$$

$$N\left[v_q\left(x + \frac{1}{N}\right) - v_q(x)\right] = v_{q-1}(x) - v_q(x)$$

We can approximate $v_q(x)$ by turning this difference equation into a differential equation

$$\frac{dv_q}{dx} + v_q = v_{q-1}$$

It can be verified that

$$v_q(x) = 1 - e^{-x} \sum_{m=0}^{q} \frac{x^m}{m!} \tag{11}$$
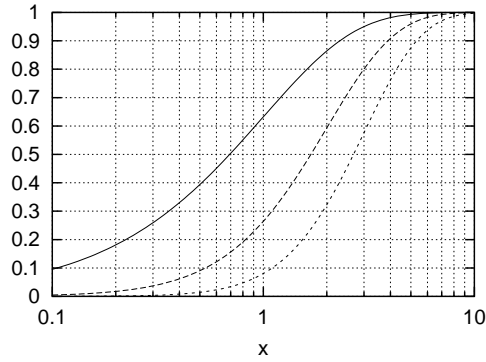
Figure 9: Curves of $v_0(x)$ (solid line), $v_1(x)$, and $v_2(x)$

is a solution to this equation. In particular, the fraction of elements that have been encountered at least twice is

$$v_1(x) = 1 - (1 + x)e^{-x} \tag{12}$$

Figure 9 shows the curves of $v_0(x)$, $v_1(x)$ and $v_2(x)$. For $x \ll 1$, set s_0 increases fast ($w_0(n) \simeq n$). Approximately, when the sequence length is two times the size of S, s_0 is almost 90% of S. For $x \ll 1$, $v_1(x) \simeq x^2$, and set s_1 is quasi empty. The sequence length must be roughly four times the size of S for s_1 to be 90% of S.

A.2 Set of k-Bit Strings

We assume S is now the set of all k-bit strings. There are $N = 2^k$ strings in S. A string is chosen from S by generating k bits with a Bernoulli process, with a probability p for a bit to be 1 and a probability $1 - p$ to be 0. We will assume $p \ll 1$ in the remaining of this section. We seek to model the growth of w_q.

The set of k-bit strings can be partitioned into $k + 1$ subsets $(S_i)_{i=0...k}$ such that all the elements in a given S_i are equiprobable : S_i is the set of strings with i bits equal to 1, its size is

$$N_i = \binom{k}{i}$$

and the probability P_i that a string belong to S_i is

$$P_i = \binom{k}{i} p^i (1 - p)^{k-i}$$
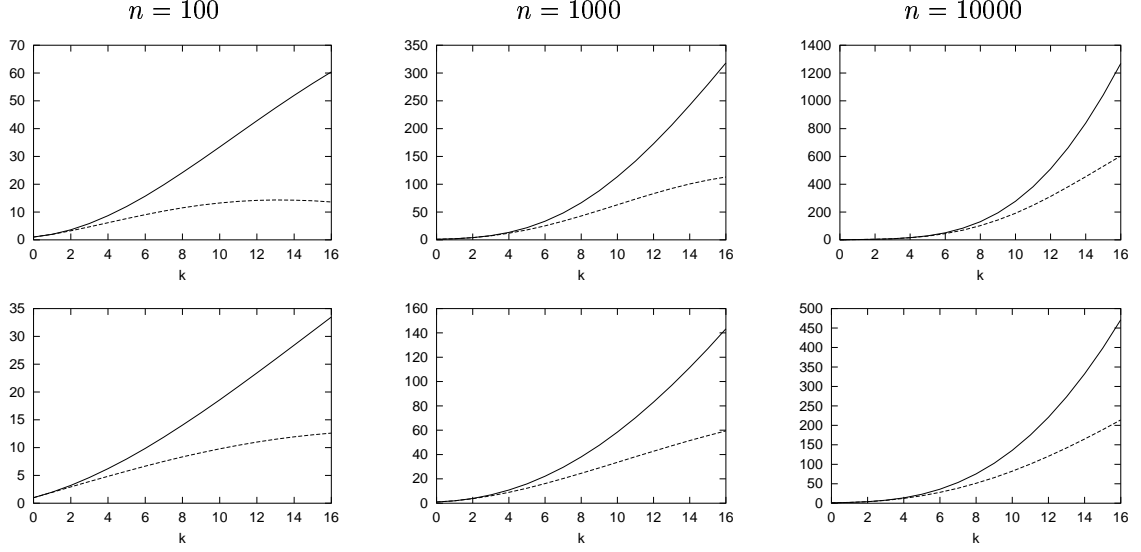
$$n = 100 \qquad\qquad n = 1000 \qquad\qquad n = 10000$$

Figure 10: Curves of $w_0(k)$ (solid line) and $w_1(k)$ (dashed line) for $n = 100$, $n = 1000$, $n = 10000$ assuming $p = 0.1$ (upper graphs) and $p = 0.05$ (lower graphs)

We apply the model of Section A.1 on each set S_i separately. In a sequence of length n, there are approximately $P_i n$ strings from S_i. Hence, defining

$$n_i = \frac{1}{p^i(1-p)^{k-i}} \simeq \left(\frac{1}{p} - 1\right)^i e^{pk} \tag{13}$$

we have

$$w_q(k, n) = \sum_{i=0}^{k} v_q(n/n_i) \times N_i \tag{14}$$

Figure 10 shows, for $p = 0.1$ and $p = 0.05$, the curves of $w_0(k)$ and $w_1(k)$ for $n = 100$, $n = 1000$ and $n = 10000$.

Some observations.   We try to understand the following observations :

- For small values of n, the curve of w_0 looks linear ($p = 0.05$, $n = 100$). As n increases, so does the the curvature, and the curve looks more like a parabola ($p = 0.05$, $n = 1000$). The curve of w_1 seems to behave similarly, but with a delay (w_1 looks linear for $p = 0.05$ and $n = 1000$, then like a parabola for $n = 10000$).

- The working set grows slower with n for smaller values of p.

- For small values of k, the curves of w_0 and w_1 are very close. They diverge beyond $k = k_{div}$, with k_{div} increasing logarithmically with n (roughly).

The first observation can be explained as follows. As p is small, sequence (n_i) increases very fast. We have $v_0(n/n_i) \approx 1$ for $n_i < n/2$ and $v_0(n/n_i) \ll 1$ for $n_i > 10n$. Similarly, $v_1(n/n_i) \approx 1$ for $n_i < n/4$ and $v_1(n/n_i) \ll 1$ for $n_i > 2n$. Hence the working set is roughly a sum of binomial coefficients

$$w_q(k, j) \approx \sum_{i=0}^{j} \binom{k}{i} \tag{15}$$

with j depending on q, n, p (and also on k when k is high, cf. Formula 13). There is no closed form for the partial sum of binomial coefficients [13], but we can have a rough approximation for small values of j because in this case, the main contribution to the sum comes from the last two coefficients

$$w_q(k, j) \approx \binom{k}{j-1} + \binom{k}{j} = \binom{k+1}{j} \tag{16}$$

For a fixed j, this is a polynomial of k of degree j. When $j = 1$, $w_q(k)$ looks linear, then when n is multiplied by $1/p$, $j = 2$ and w_q looks like a parabola.

The second observation is closely related to the first one. From Formula 13, the value of j may be estimated as

$$j = \frac{\log(n/x_q) - pk}{\log(1/p - 1)} \tag{17}$$

Parameter x_q depends on q. For $q = 0$, we found that $x_0 = 1.5$ associated with the approximation of Formula 16 gives good results for large values of n (when j is not an integer, we used the gamma function). For $q = 1$, we found $x_1 = 4.5$ to give good results (the higher n, the better). In general pk is negligible compared with $\log(n/x_q)$. When we decrease p, we increase j, which explains the second observation. Note that when $j > 1$, a variation of p (e.g., a division by 2) has more effect on j than a variation of n (e.g, a multiplication by 2).

The third observation can be explained as follows. As $v_0(x)$ increases faster than $v_1(x)$ (Figure 9), when n is large enough for set S_j to be almost totally included in s_1, s_0 already contains a non-negligible number of elements from set S_{j+1}. As long as $j < k/2$, $\binom{k}{j+1}$ is significant compared with $\sum_{i=0}^{j} \binom{k}{i}$, and the difference between w_0 and w_1 is non negligible. When $j \geq k/2$, this is no longer the case. Hence we have approximately $k_{div} \approx 2j$.
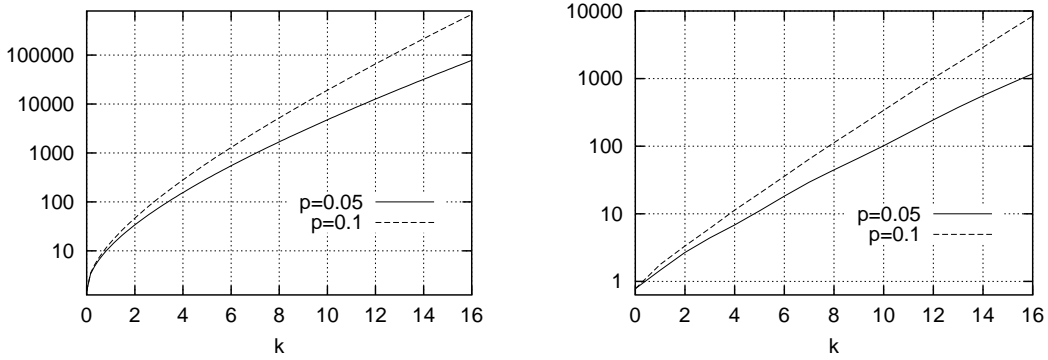
Figure 11: Left graph : value $n = n_w$ beyond which s_0 increases slowly with n. Right graph : useful working set $w_0(n_w, k)$

The "useful" working set.   Let us consider the following sum

$$\sum_{i=0}^{j} P_i = \sum_{i=0}^{j} \binom{k}{i} p^i (1-p)^{k-i}$$

This sum is approximately the probability that a string belongs to the working set already gathered. When this sum is close to 1, we can consider that the "useful" working set is gathered. Of course, the working set will continue to grow, but very slowly.

We search the approximate value $n = n_w$ beyond which $\sum_{i=0}^{j} P_i$ is close to 1. This sum is the distribution function of a binomial distribution. This binomial distribution can be approximated by a normal distribution centered on kp and with standard deviation $\sigma = \sqrt{kp(1-p)}$. The area under a gaussian is negligible after 3σ, hence we can consider that the working set is gathered when

$$j = kp + 3\sqrt{kp(1-p)}$$

that is

$$\log(n_w) = \left(kp + 3\sqrt{kp(1-p)} \right) \log\left(\frac{1}{p} - 1 \right) + pk + \log(x_0)$$

The left graph on Figure 11 shows the value of $n_w(k)$ on a $\log_{10}$ scale for $p = 0.05$ and $p = 0.1$. The "time" necessary to gather the useful working set grows (roughly) exponentially with k. Using Formula 14, we verified that, for $n > n_w$, the instantaneous miss ratio $w_0(n+1, k) - w_0(n, k)$ is less than 1%.

The right graph on Figure 11 shows the useful working set $w_0(n_w, k)$. The useful working set increases (roughly) exponentially with k. However, its size also depends on p. The useful

working set is larger for higher values of p. It can be noticed that for $k = 16$, going from $p = 0.05$ to $p = 0.1$ incurs a ten fold increase of the useful working set.

B   Experimental Results

benchmark	# dyn br	# inst/dyn br	# stat br	90 %	% misp 2bc
IBS					
groff	1.16×10^7	7.5	5633	437	4.4
gs	1.43×10^7	7.1	10933	932	6.6
mpeg_play	8.11×10^6	10.2	4752	419	8.1
nroff	2.14×10^7	4.9	4480	199	4.0
half real_gcc	6.88×10^6	6.4	11551	2107	8.0
real_gcc	1.39×10^7	6.6	16712	3021	9.5
sdet	5.22×10^6	6.2	4583	376	4.7
verilog	5.69×10^6	7.3	3916	673	6.7
video_play	5.18×10^6	8.5	3977	597	6.4
from **SPEC CPU95**					
go	1.60×10^7	8.3	5102	1092	20.9
from **SPEC CPU2000**					
mcf	3.23×10^7	5.4	844	42	12.3
twolf	3.22×10^7	6.7	2652	177	22.3
gap	1.27×10^8	7.3	4081	421	7.1

Table 1: Benchmarks statistics. Number of conditional branches executed, number of dynamic instructions per conditional branch executed, number of unique static branches encountered, number of static branches that contributed to 90% of conditional branches executed, percentage of conditional branches mispredicted with two-bit counters (one for each static branch).
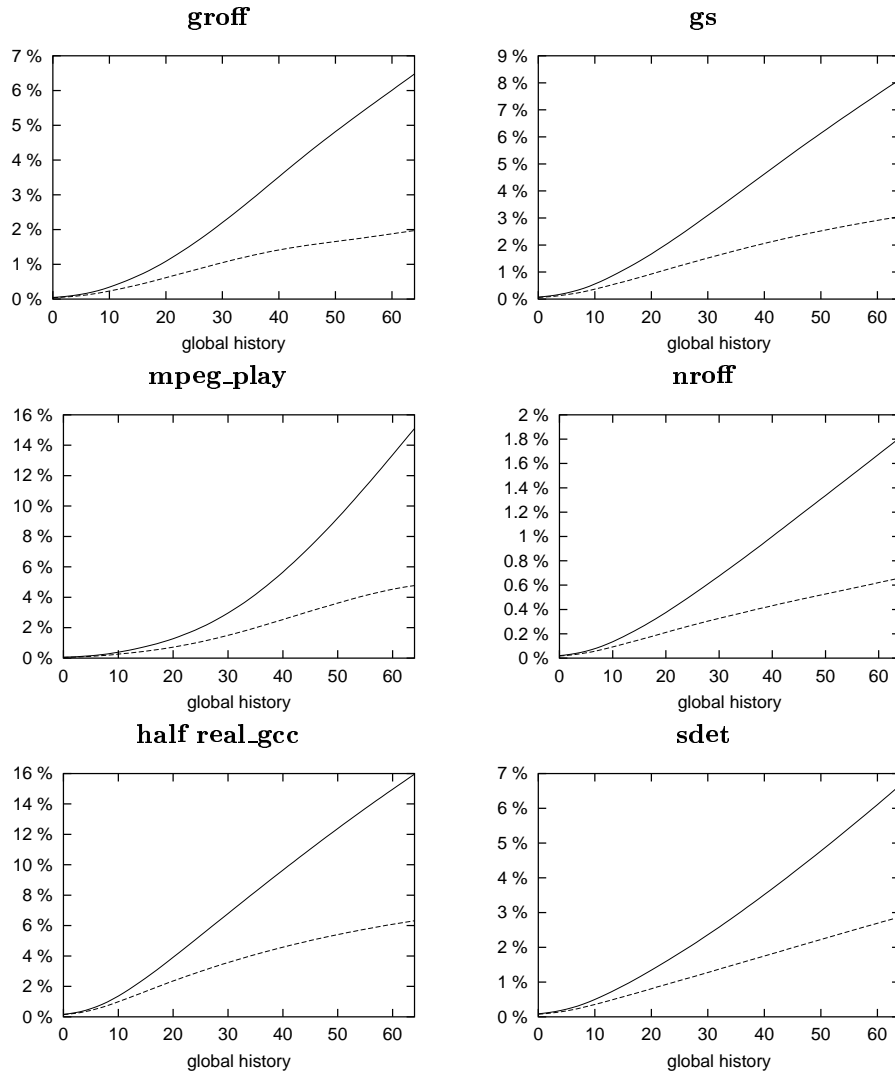
Figure 12: BPPM working set. The solid curve represents the number W_h of distinct (B, H_h) pairs that were encountered at least once. The dashed curve represents the number of distinct (B, H_h) pairs that were encountered at least twice. Values on the y-axis are expressed as a percentage of dynamic branches (the solid curve can be viewed as a miss ratio).

verilog

video_play

go

mcf

twolf

gap
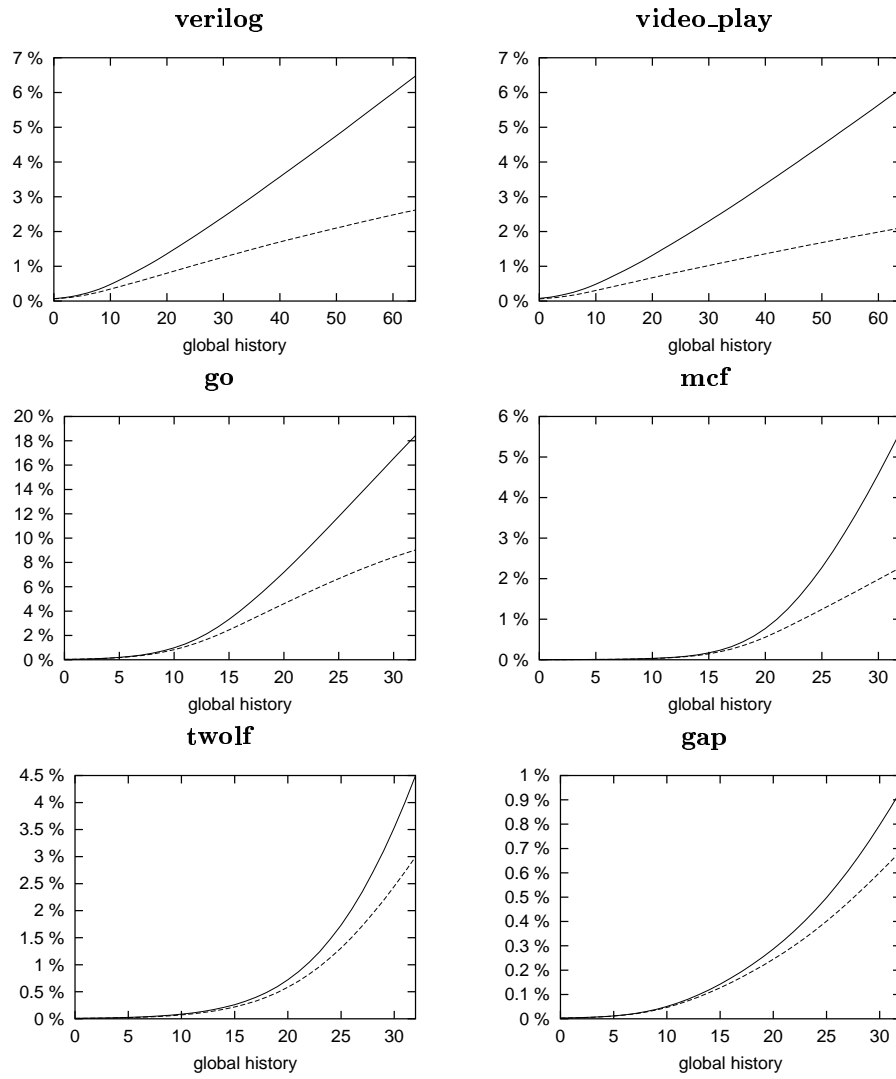
Figure 13: BPPM working set, Cf. Figure 12. The global history length is limited to 32 on the four SPEC benchmarks.

groff

gs

mpeg_play

nroff

half real_gcc

sdet
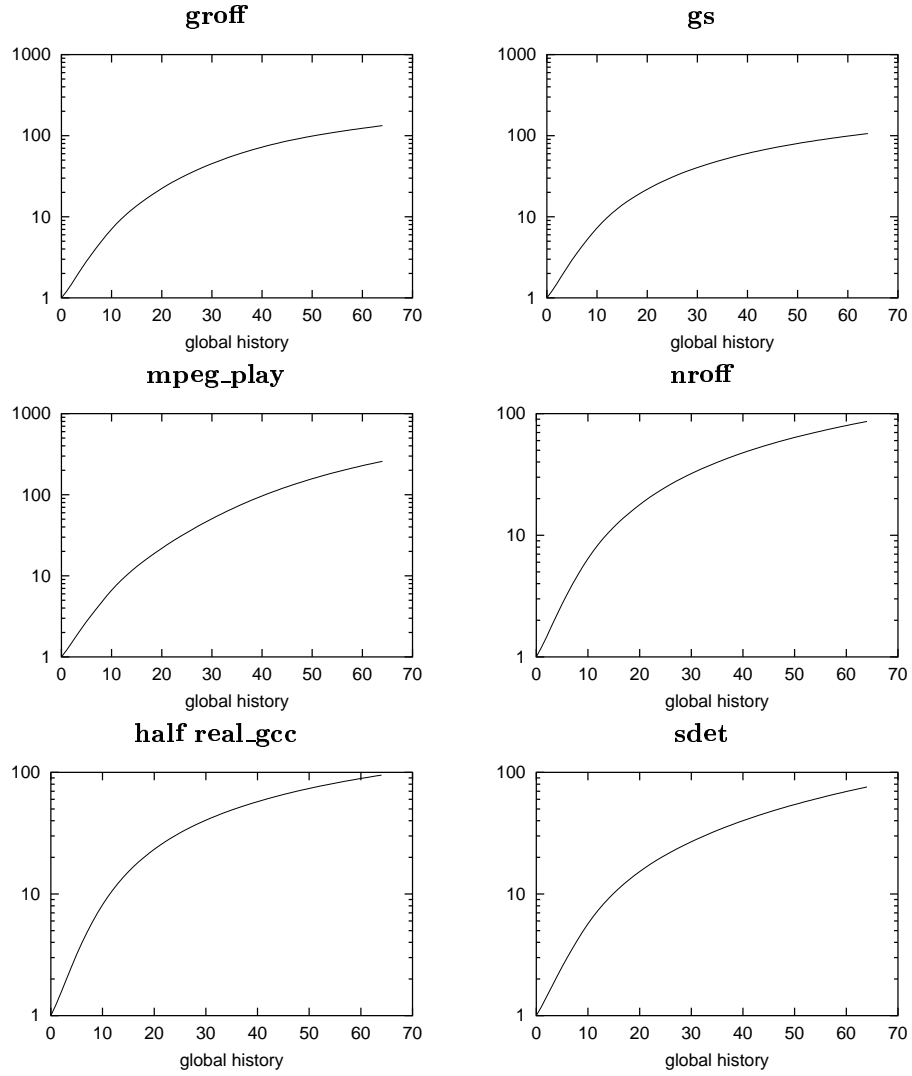
Figure 14: BPPM working set : W_h/W_0 on a logarithmic scale, as a function of h.
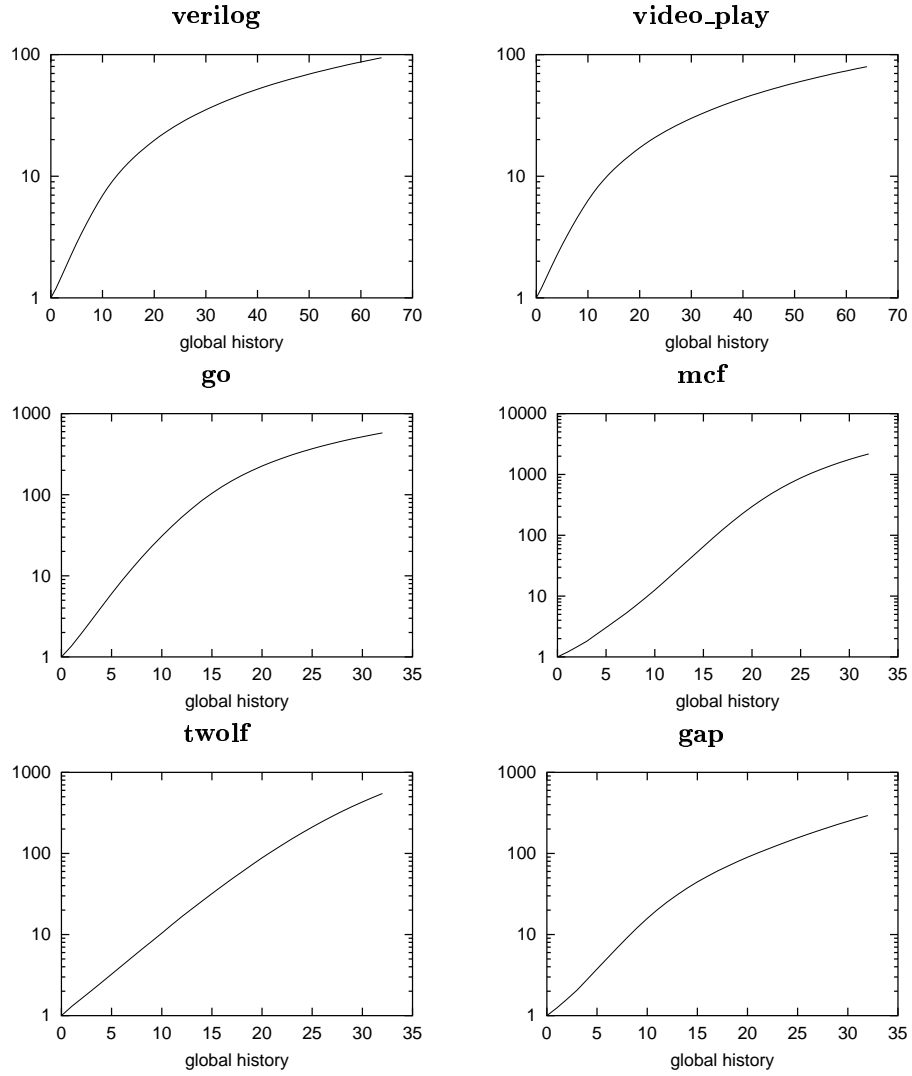
Figure 15: BPPM working set : W_h/W_0 on a logarithmic scale, as a function of h.
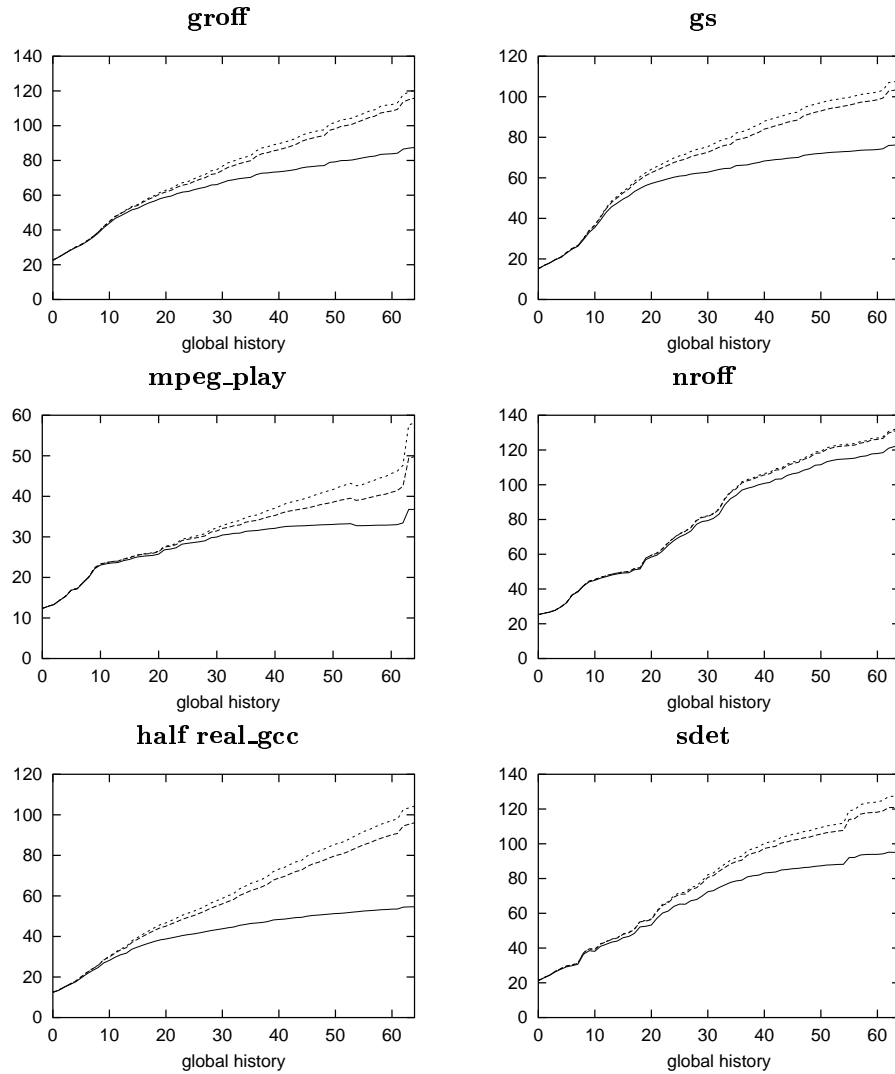
Figure 16: BPPM misprediction interval. Represented on the y-axis is the number of dynamic conditional branches per branch misprediction (higher is better). The solid curve represents overall mispredictions. The dashed curves represent the inverse of the misprediction probability on (B, H_h) pairs that were already encountered at least one time and three times respectively.
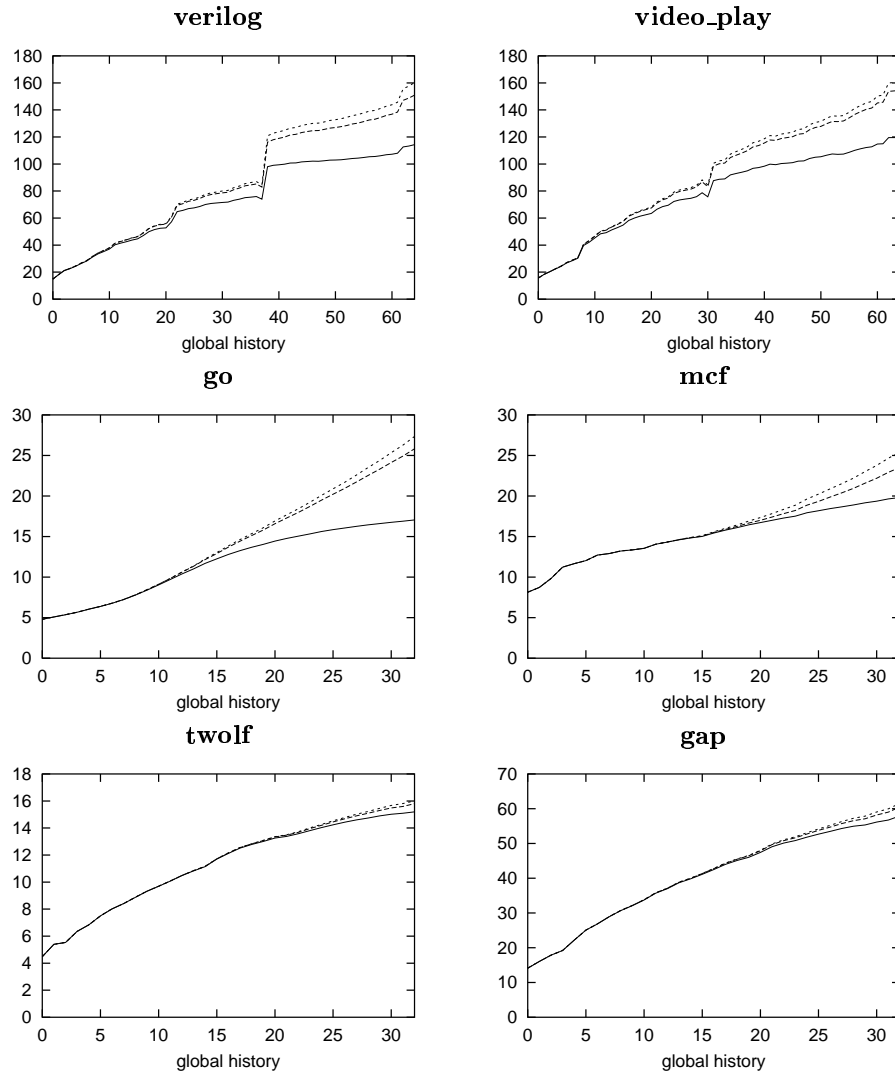
Figure 17: BPPM misprediction interval, cf. Figure 16.
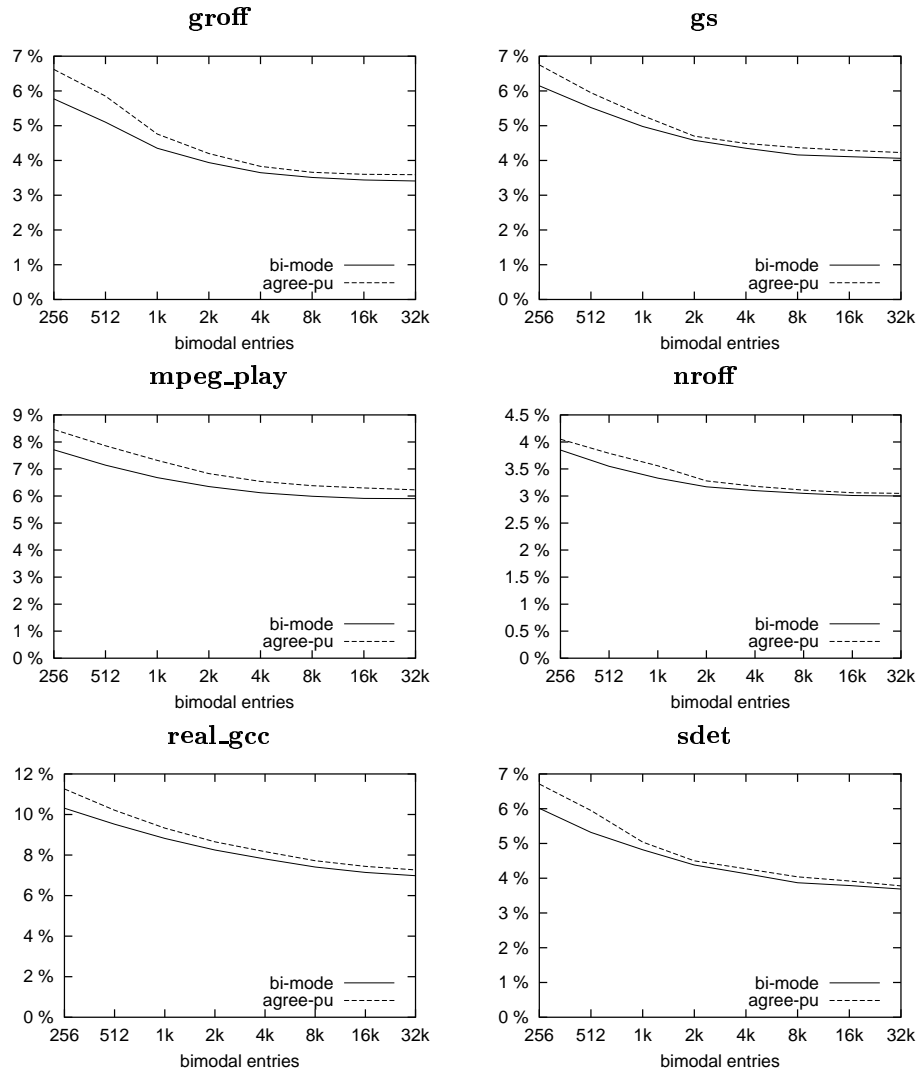
Figure 18: Comparison of bias-split and bias-agree. Misprediction percentage of bi-mode and agree-pu predictors which gshare component is fixed (4k entries, $h = 10$) and bimodal component is varied from 256 to 32k entries.

Figure 19: Cf. Figure 18.

groff

gs

mpeg_play

nroff

real_gcc

sdet
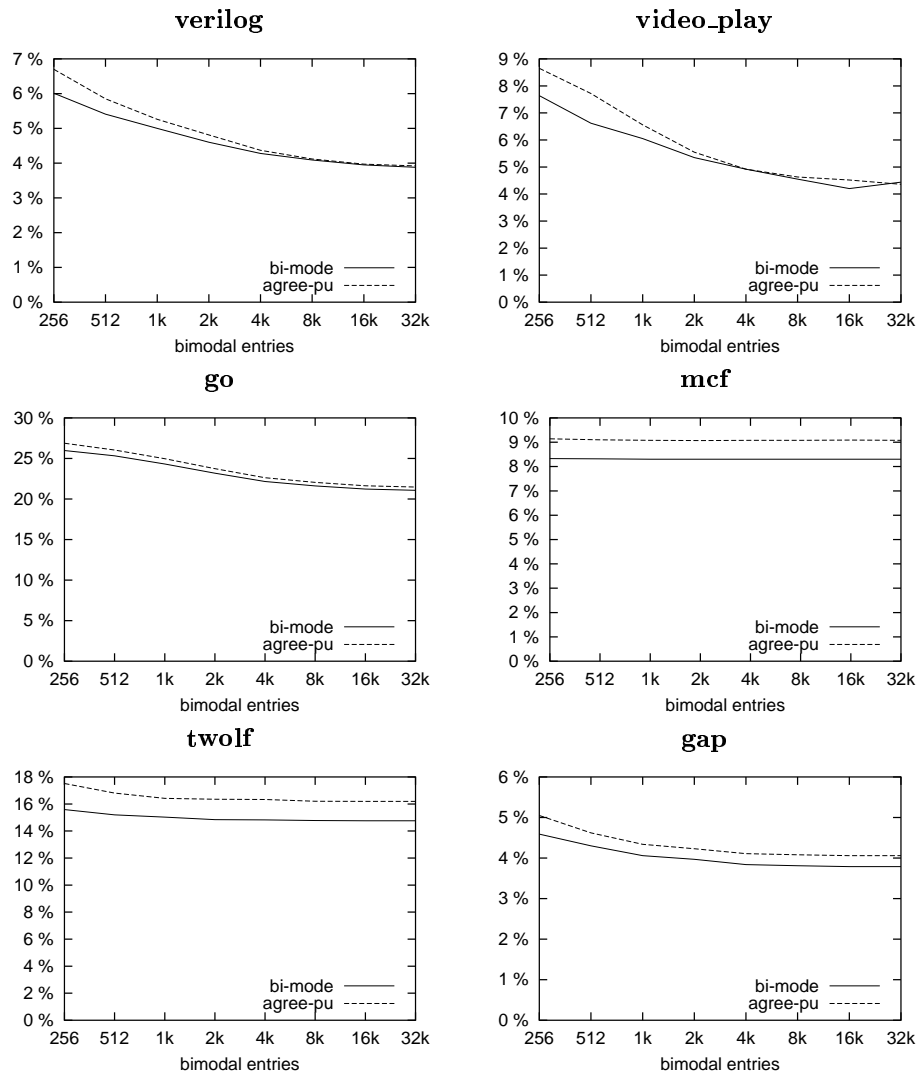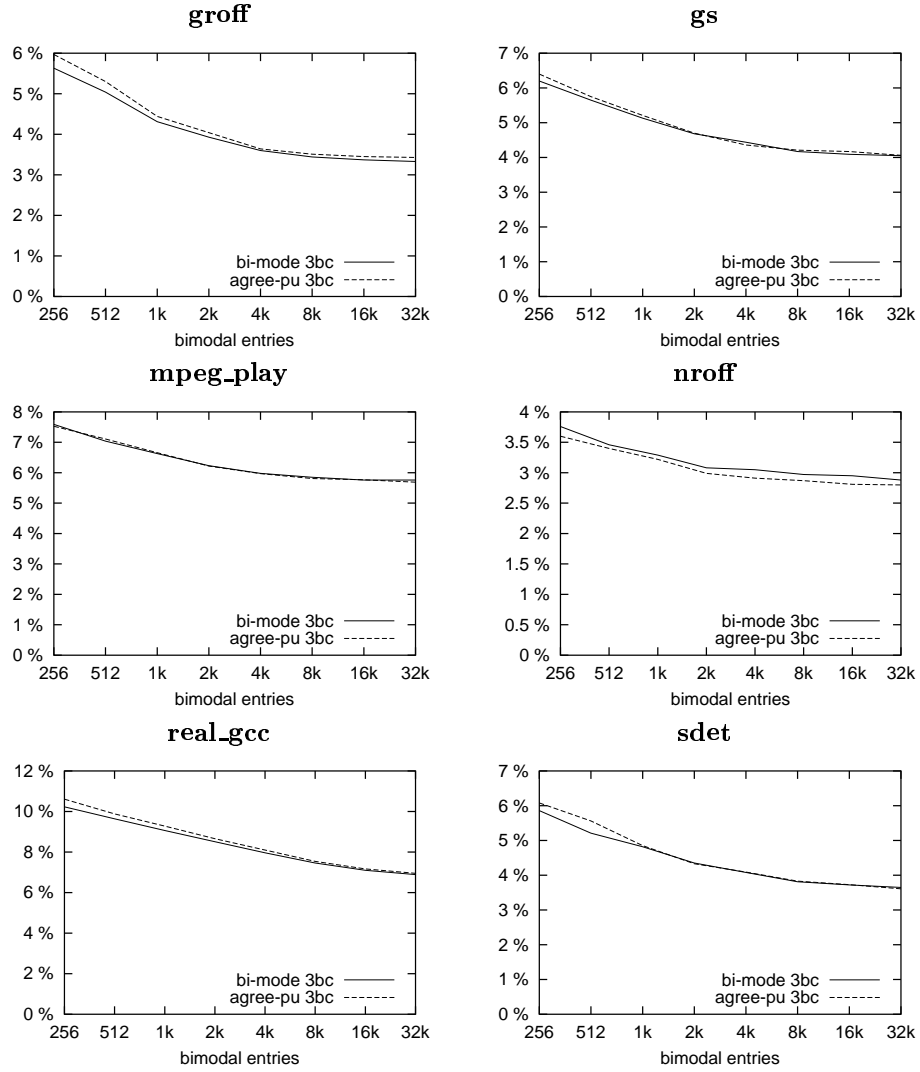
Figure 20: Same experiment as on Figure 18, but two-bit counters in the bimodal predictor have been replaced with three-bit counters.

verilog

video_play

go

mcf

twolf

gap

Figure 21: Cf. Figure 20.

groff

gs

mpeg_play

nroff

real_gcc

sdet
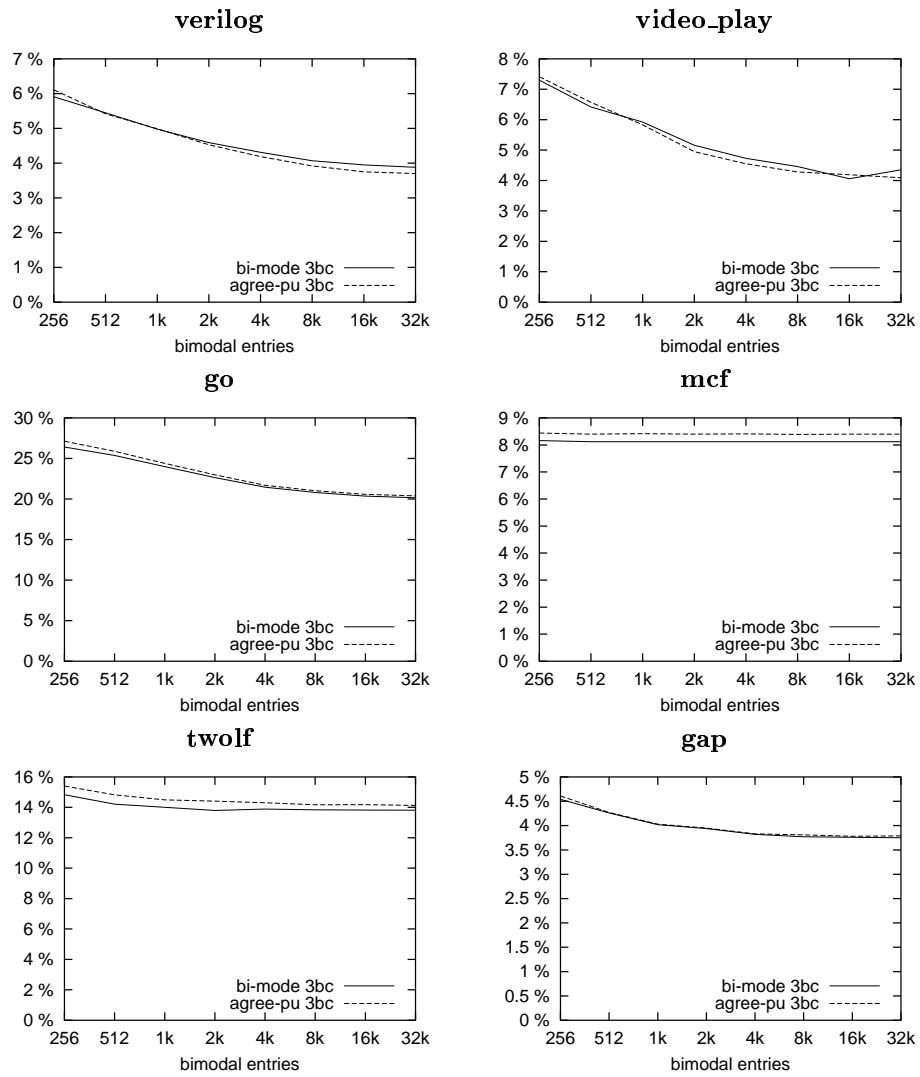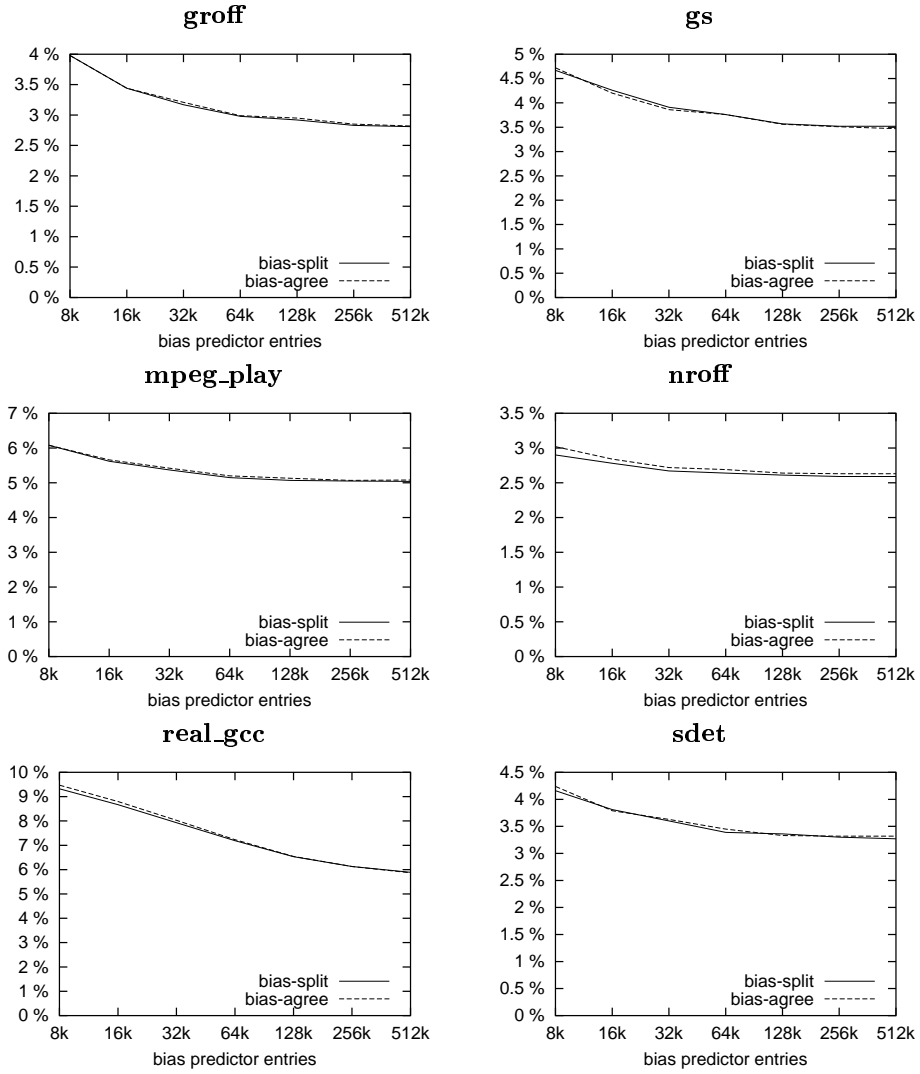
Figure 22: Same experiment as on Figure 18 but the bimodal table has been replaced with a $gshare[A, H_{10}, m]$ table, with m varied from 13 to 19.
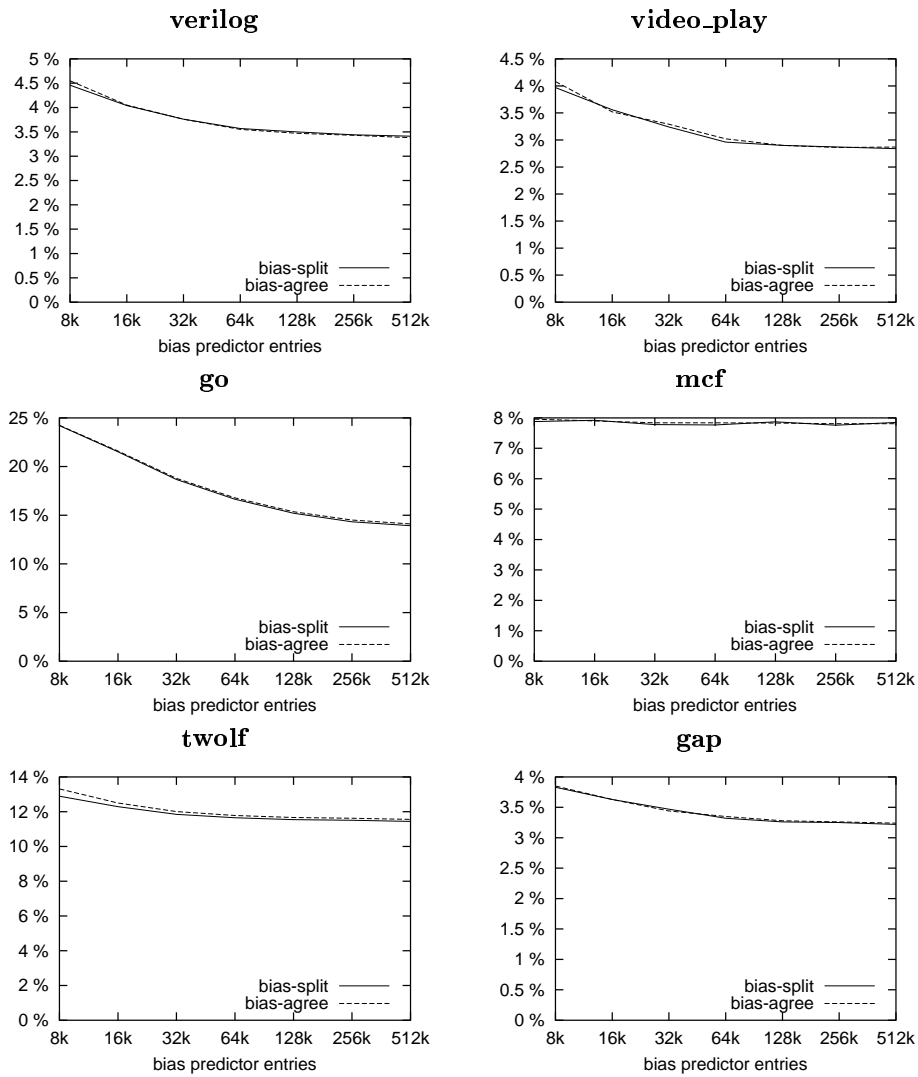
Figure 23: Cf. Figure 22.
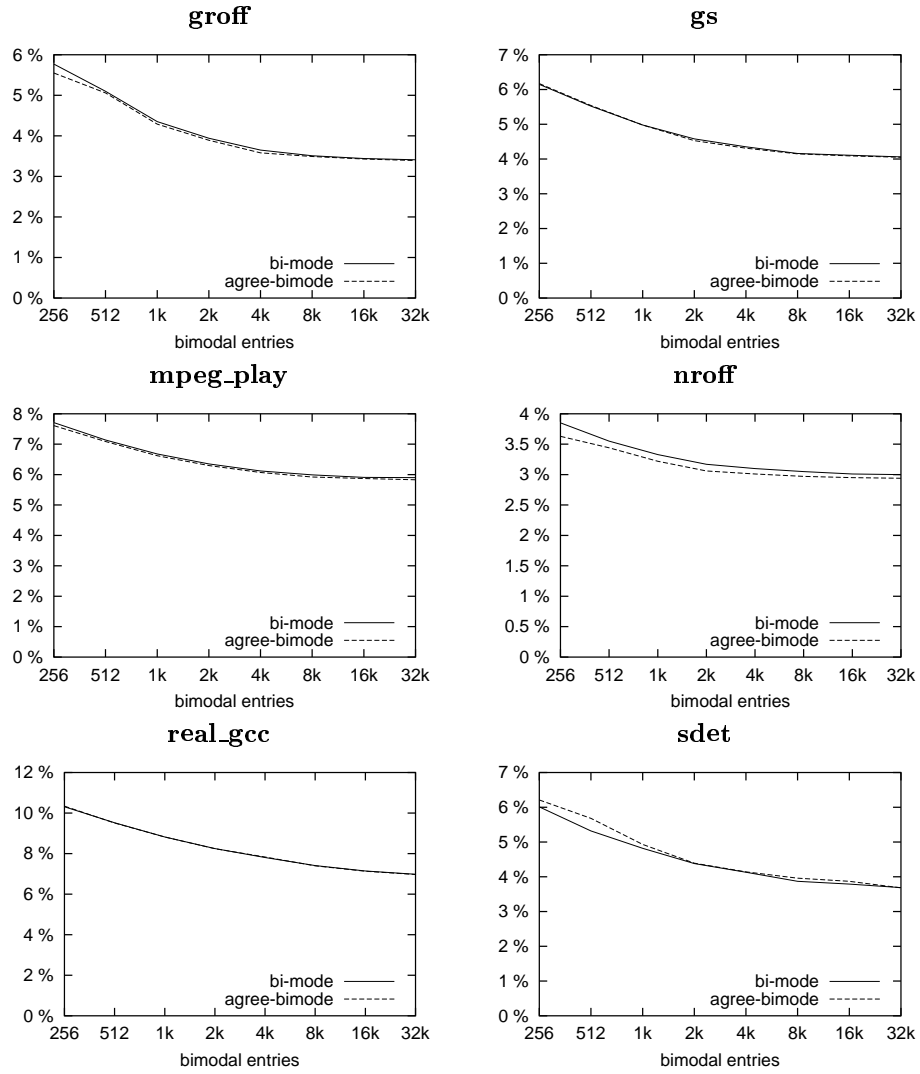
Figure 24: Comparison of bias-split and bias-inject. Misprediction percentage of bi-mode and agree-bimode predictors which gshare table is fixed (4k entries, $h = 10$) and bimodal table is varied from 256 to 32k entries.

Figure 25: Cf. Figure 24.

groff

gs

mpeg_play

nroff

real_gcc

sdet
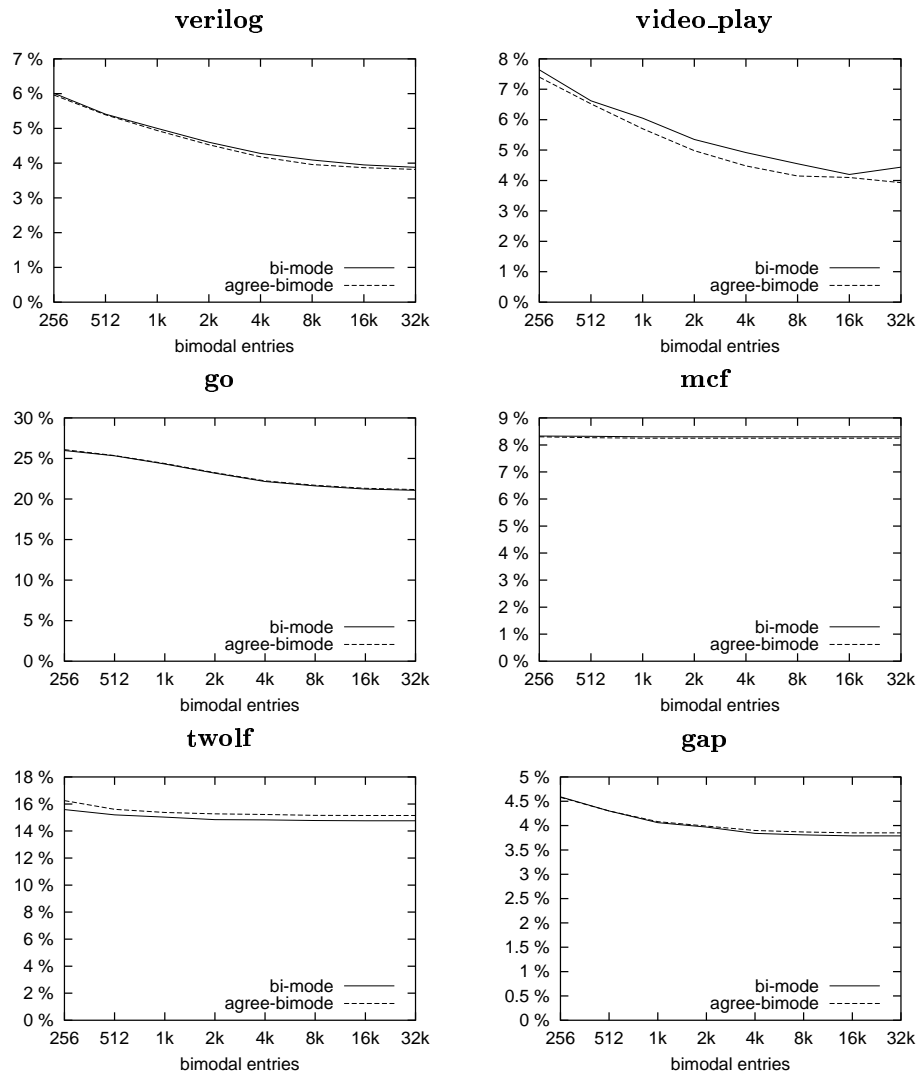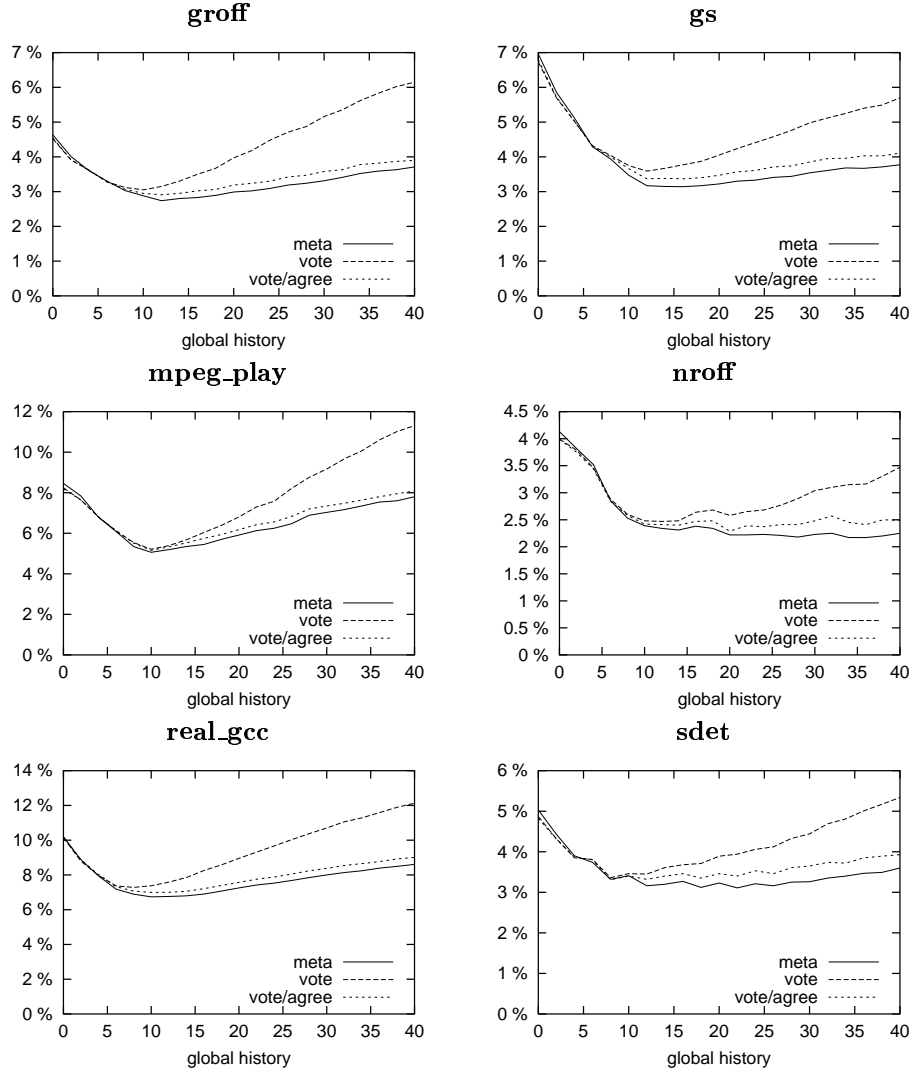
Figure 26: Misprediction percentage of *meta-select*(T_1, T_2, T_3), *majority-vote*(T_1, T_2, T_3) and *majority-vote*$(T_1, T_2, bias\text{-}agree(T_1, T_3))$, with $T_1 = bimodal[A, 12]$, $T_2 = gshare1[A, H_h, 12]$ and $T_3 = gshare2[A, H_h, 12]$

verilog

video_play

go

mcf

twolf

gap
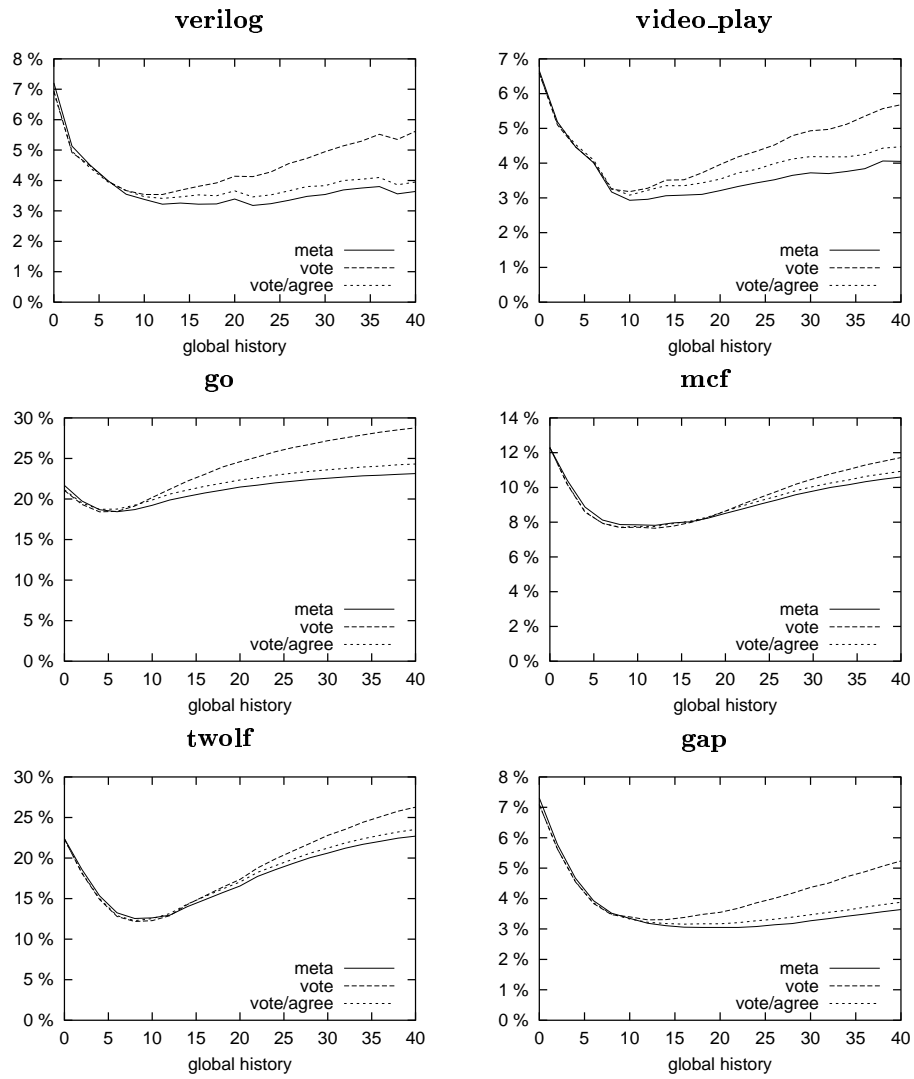
Figure 27: Cf. Figure 26.
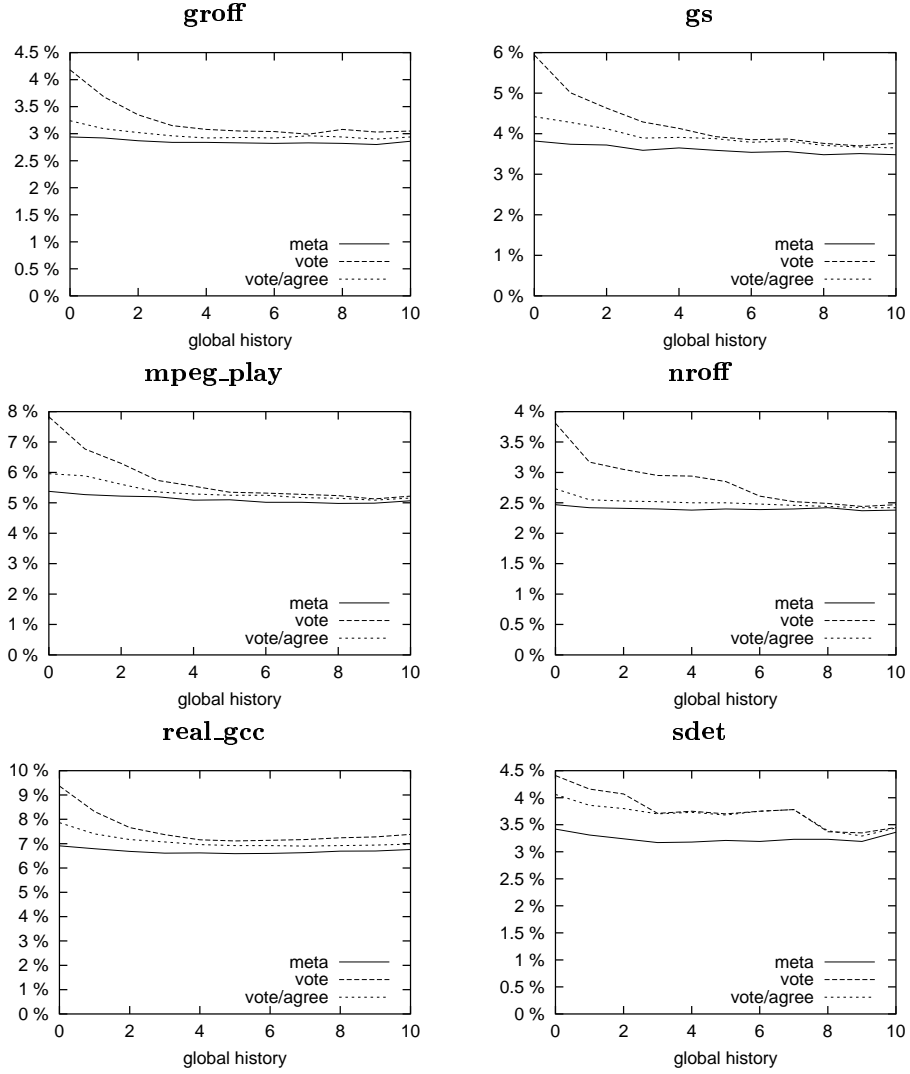
Figure 28: Misprediction percentage of $meta\text{-}select(T_1, T_2, T_3)$, $majority\text{-}vote(T_1, T_2, T_3)$ and $majority\text{-}vote(T_1, T_2, bias\text{-}agree(T_1, T_3))$, with $T_1 = bimodal[A, 12]$, $T_2 = gshare1[A, H_{10}, 12]$ and $T_3 = gshare2[A, H_h, 12]$, with h varied from 0 to 10

Figure 29: Cf. Figure 28.

groff

gs

mpeg_play

nroff

real_gcc

sdet
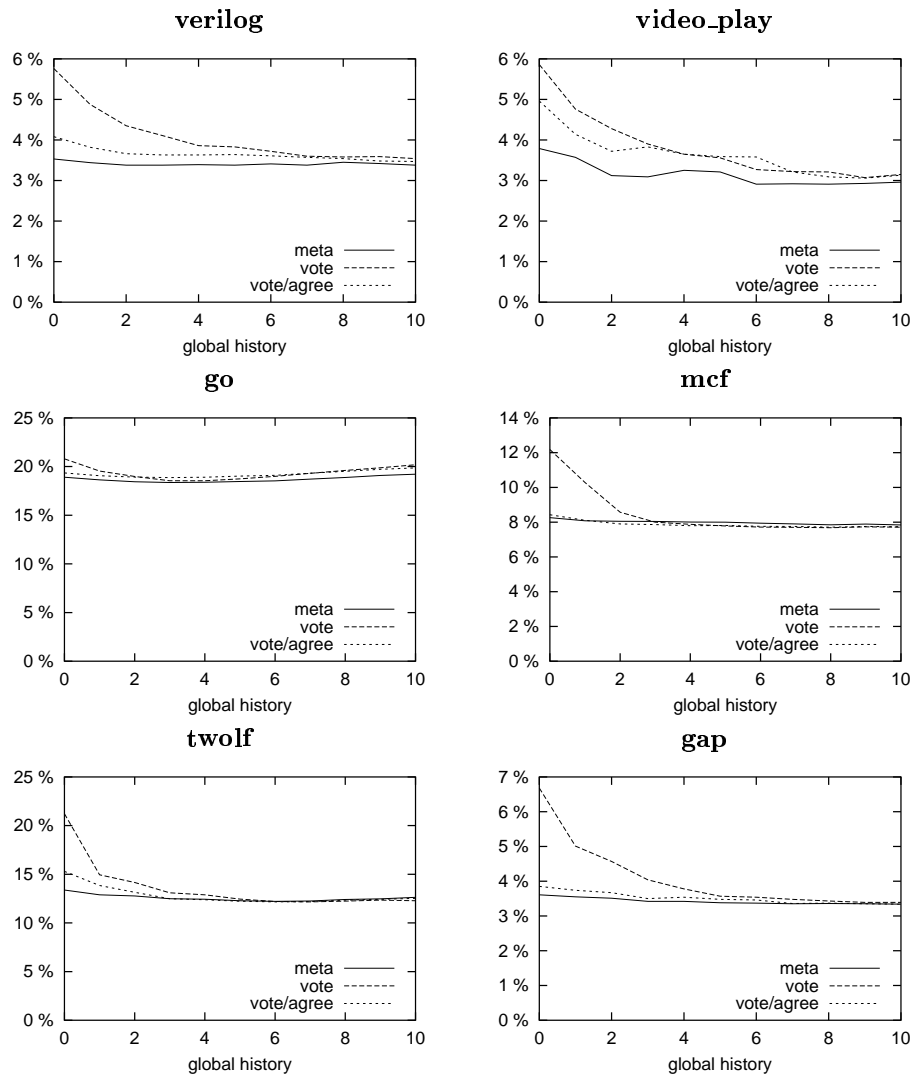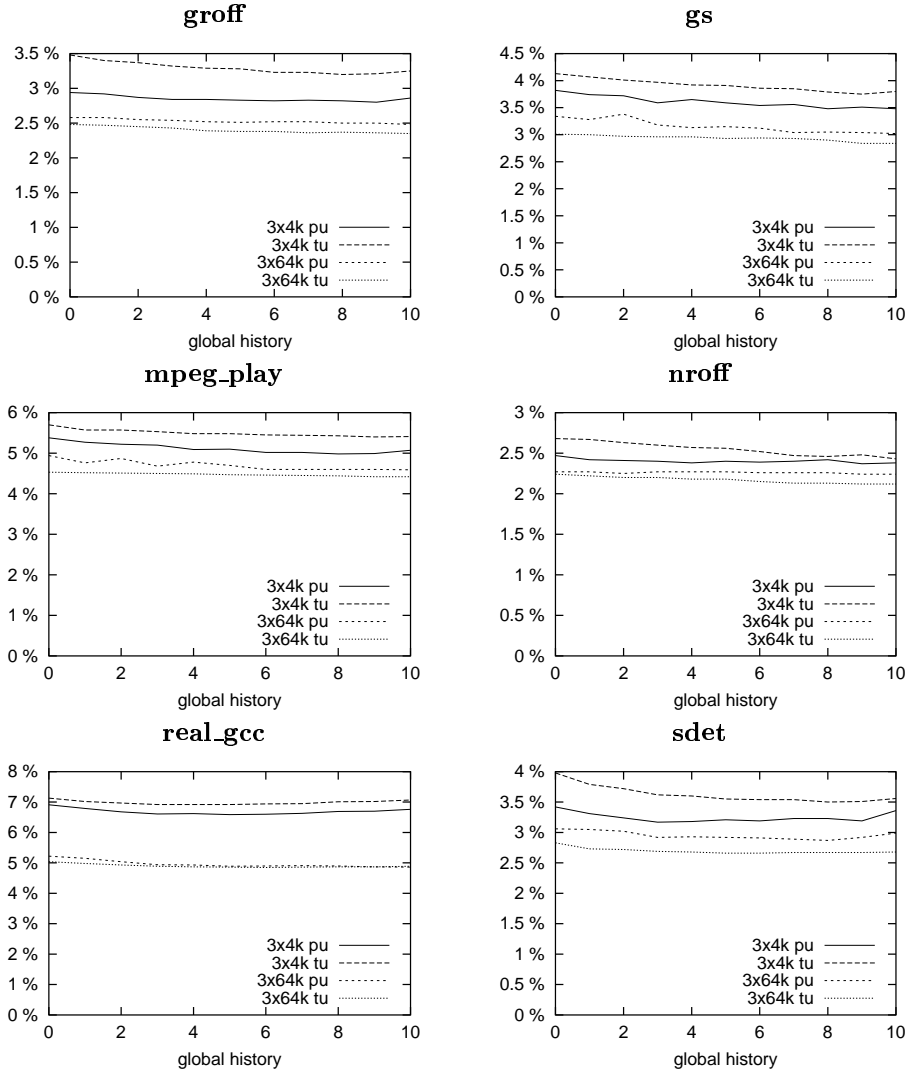
Figure 30: Impact on prediction accuracy of partial update ("pu") vs. total update ("tu") on *meta-select*(T_1, T_2, T_3), with $T_1 = bimodal[A, m]$, $T_2 = gshare1[A, H_{10}, m]$ and $T_3 = gshare2[A, H_h, m]$, with h varied from 0 to 10, and for $m = 12$ (3x4k) and $m = 16$ (3x64k)

Figure 31: Cf. Figure 30.

groff

gs

mpeg_play

nroff

real_gcc

sdet
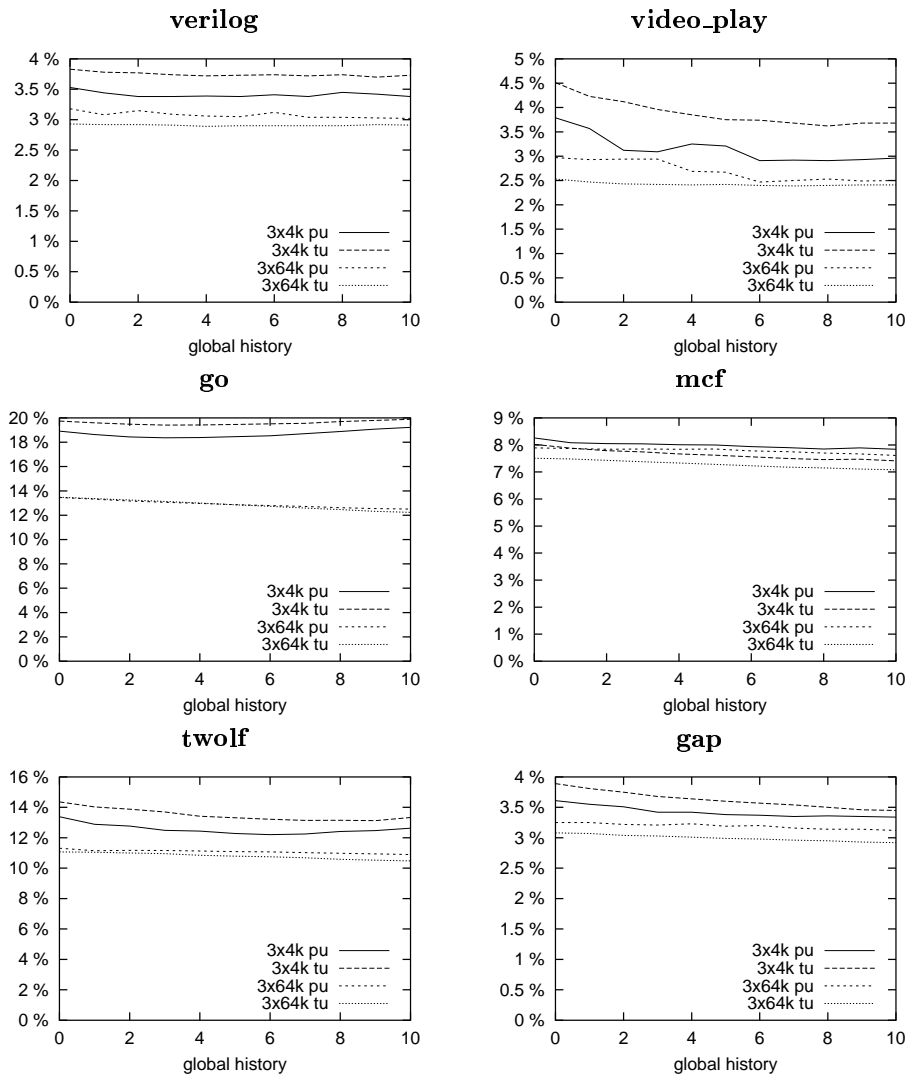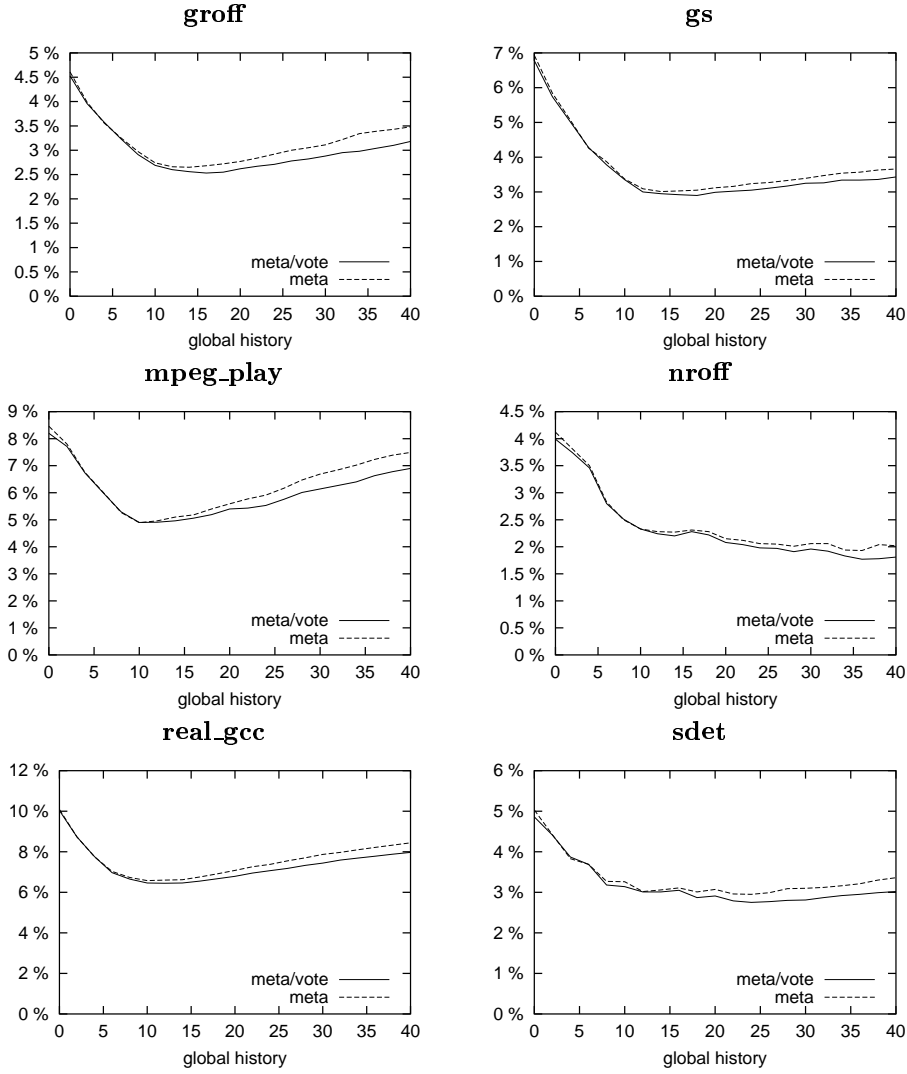
Figure 32: Misprediction percentage of a 4-KByte 2bc-gskew ("meta/vote") and a simple 4-KByte hybrid predictor ("meta") . More precisely, the 2bc-gskew predictor is $meta\text{-}select(majority\text{-}vote(T_1, T_2, T_3), T_1, T_{meta})$ with $T_1 = bimodal[A, 12]$, $T_2 = gshare1[A, H_h, 12]$, $T_3 = gshare2[A, H_h, 12]$ and $T_{meta} = gshare3[A, H_h, 12]$. The hybrid predictor is $meta\text{-}select(T_1, T'_2, T_3)$ with $T'_2 = gshare1[A, H_h, 13]$.

verilog

video_play

go

mcf

twolf

gap

Figure 33: Cf. Figure 32.

groff

gs

mpeg_play

nroff

real_gcc

sdet
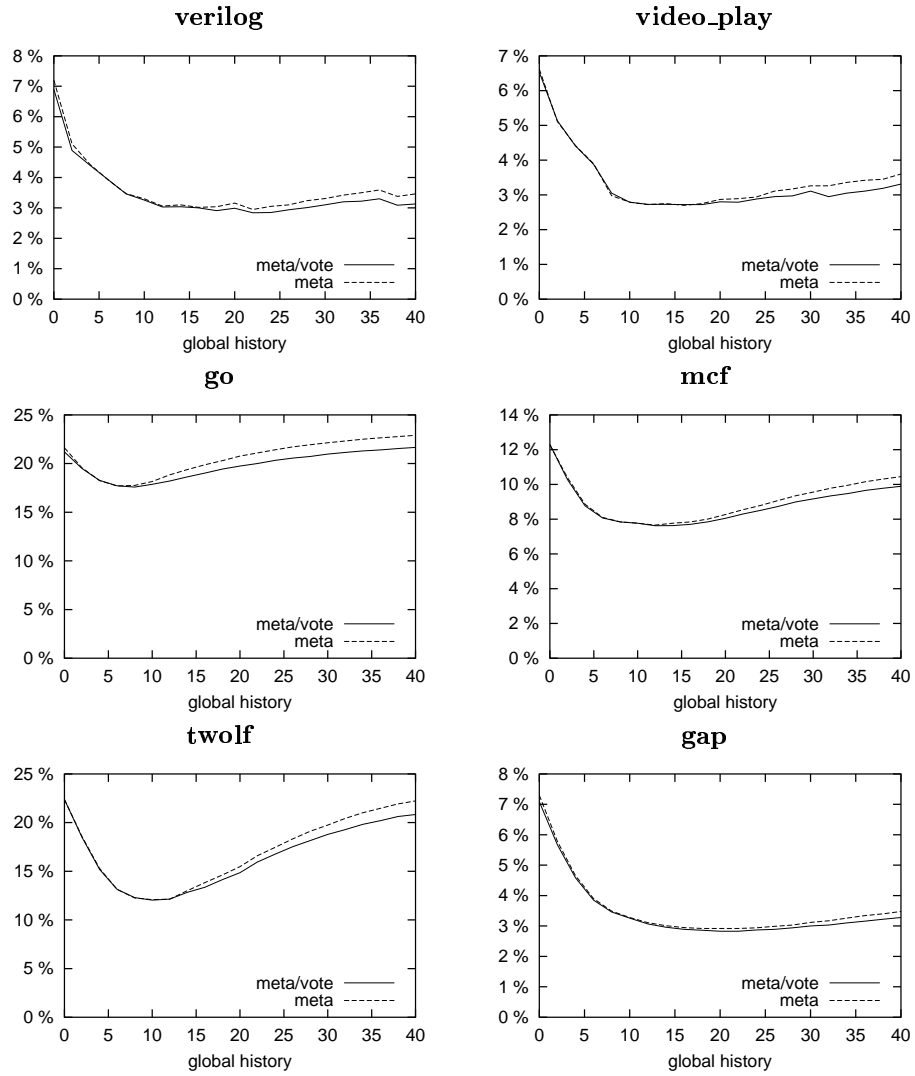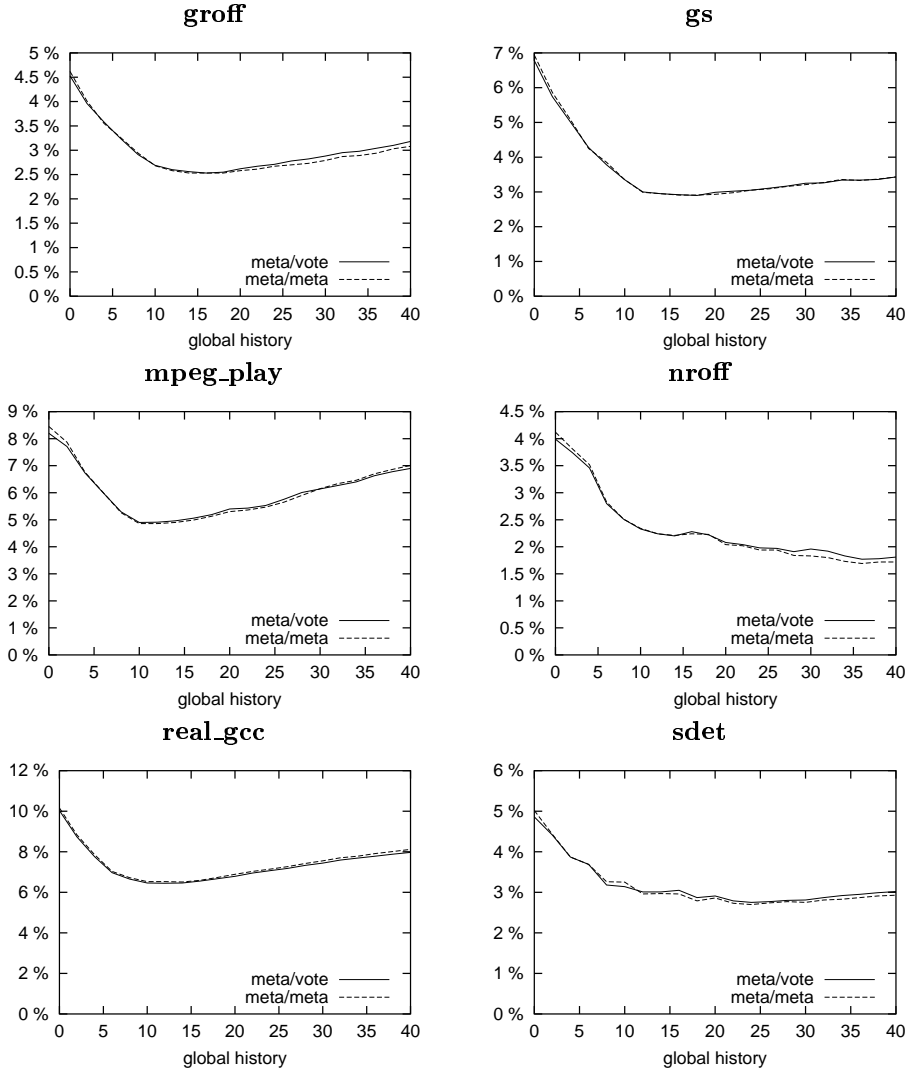
Figure 34: Misprediction percentage of a 4-KByte 2bc-gskew ("meta/vote") predictor defined as $meta\text{-}select(majority\text{-}vote(T_1, T_2, T_3), T_1, T_{meta})$ with $T_1 = bimodal[A, 12]$, $T_2 = gshare1[A, H_h, 12]$, $T_3 = gshare2[A, H_h, 12]$ and $T_{meta} = gshare3[A, H_h, 12]$. We show the impact of replacing the majority-vote by a meta-select ("meta/meta") : $meta\text{-}select(meta\text{-}select(T_1, T_2, T_3), T_1, T_{meta})$.

verilog

video_play

go

mcf

twolf

gap
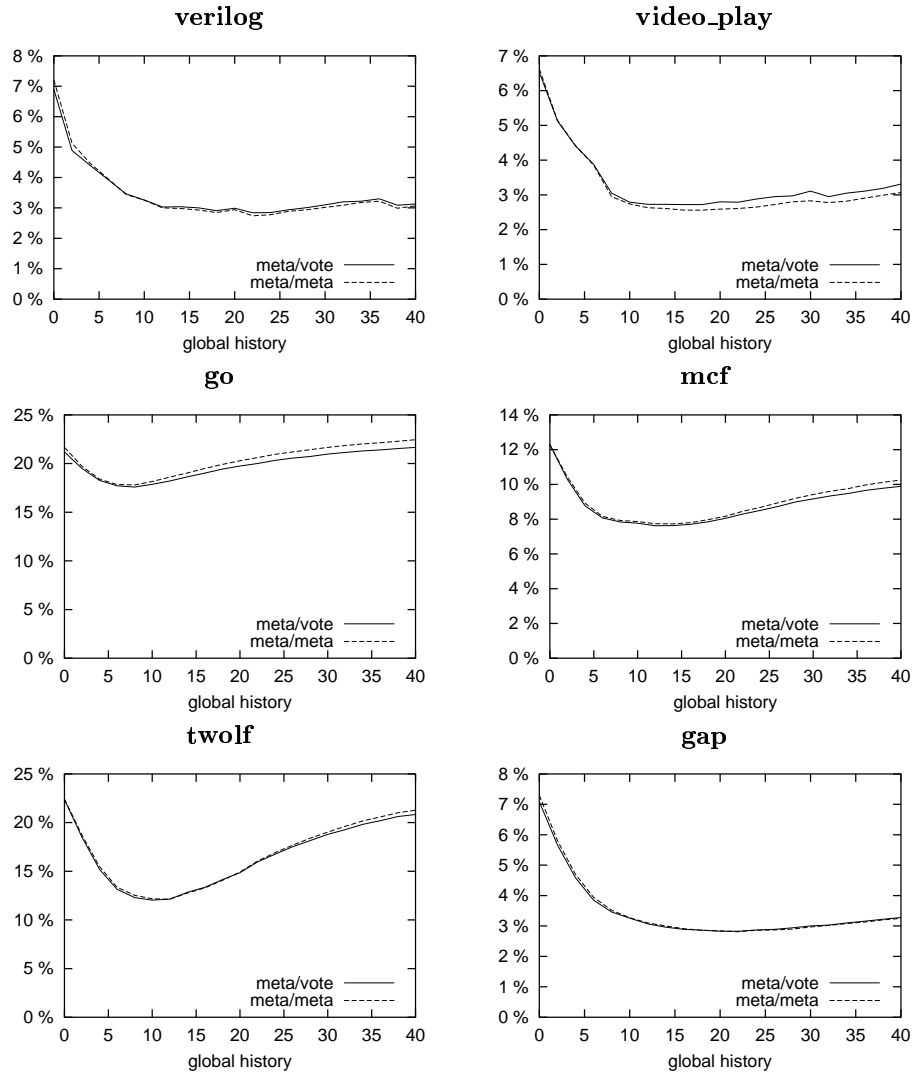
Figure 35: Cf. Figure 34.
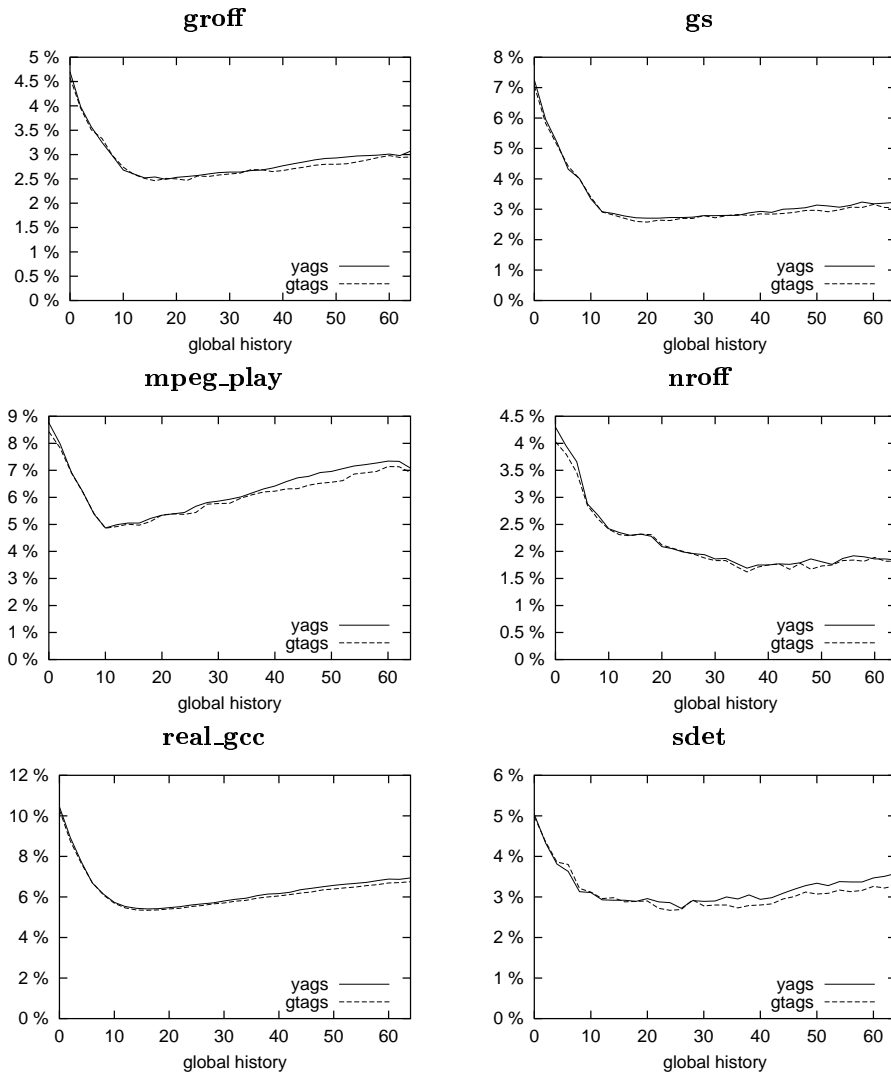
Figure 36: Misprediction percentage of $YAGS6[h, 12]$ and $gtags6[h, 12]$ for a global history length h varying from 0 to 64 bits.

Figure 37: Cf. Figure 36.

groff

gs

mpeg_play

nroff

real_gcc

sdet
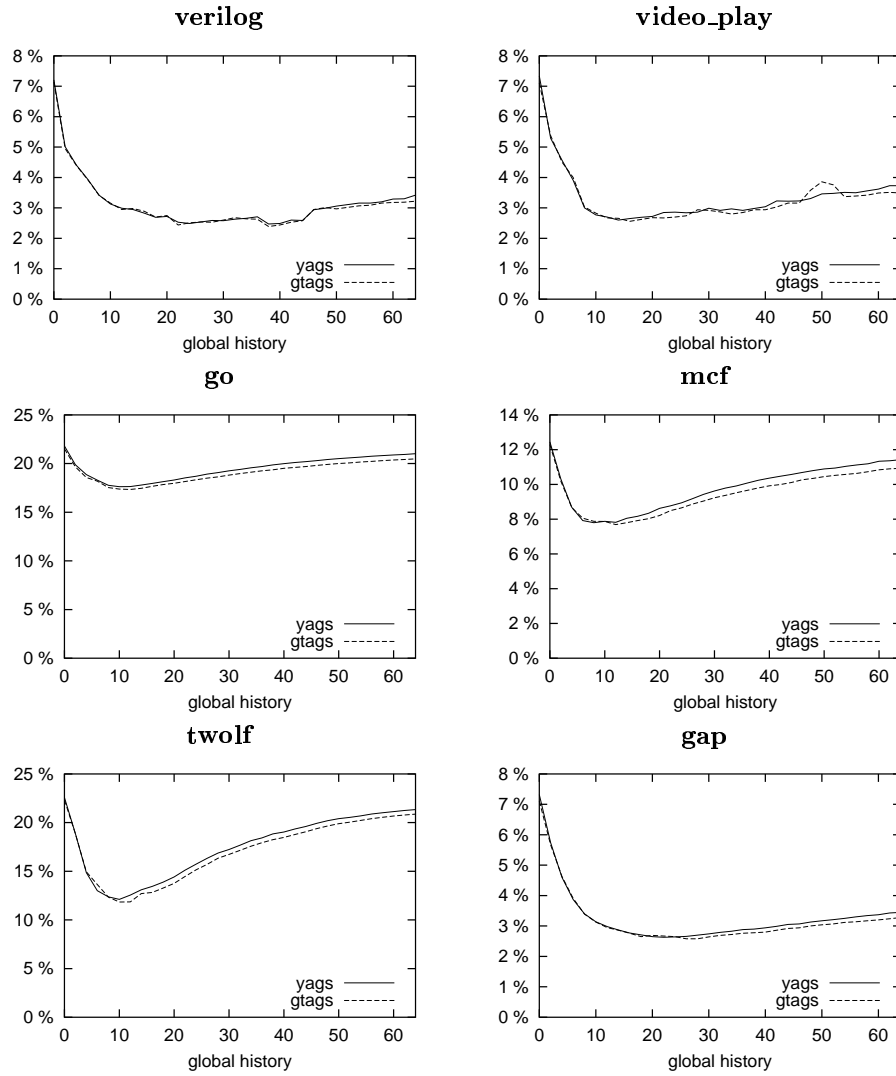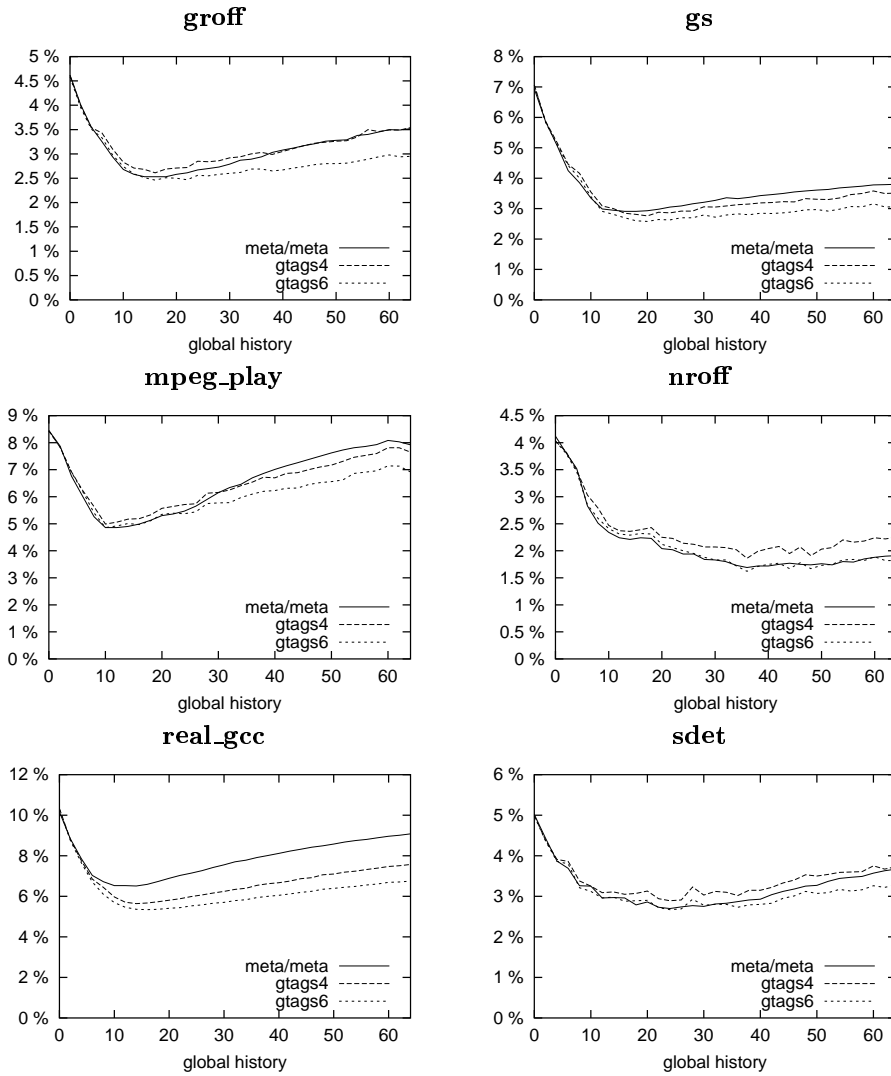
Figure 38: Comparison of gtags4 and meta/meta for a 4 KB budget. Both predictors feature the same quantity of "bimodal" prediction, "gshare" prediction and "selection" information. We also show the misprediction percentage of a gtags6 (5 KB).

verilog

video_play

go

mcf

twolf

gap

Figure 39: Cf. Figure 38.

groff

gs

mpeg_play

nroff

real_gcc

sdet
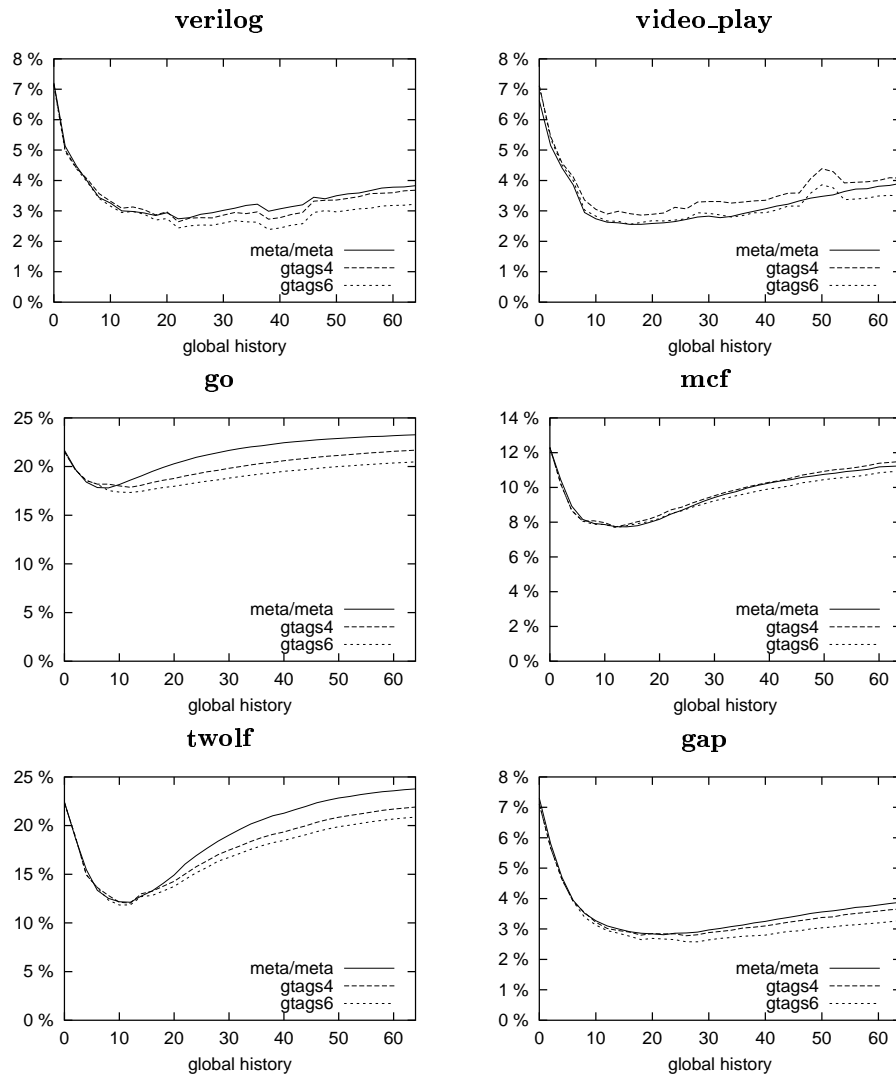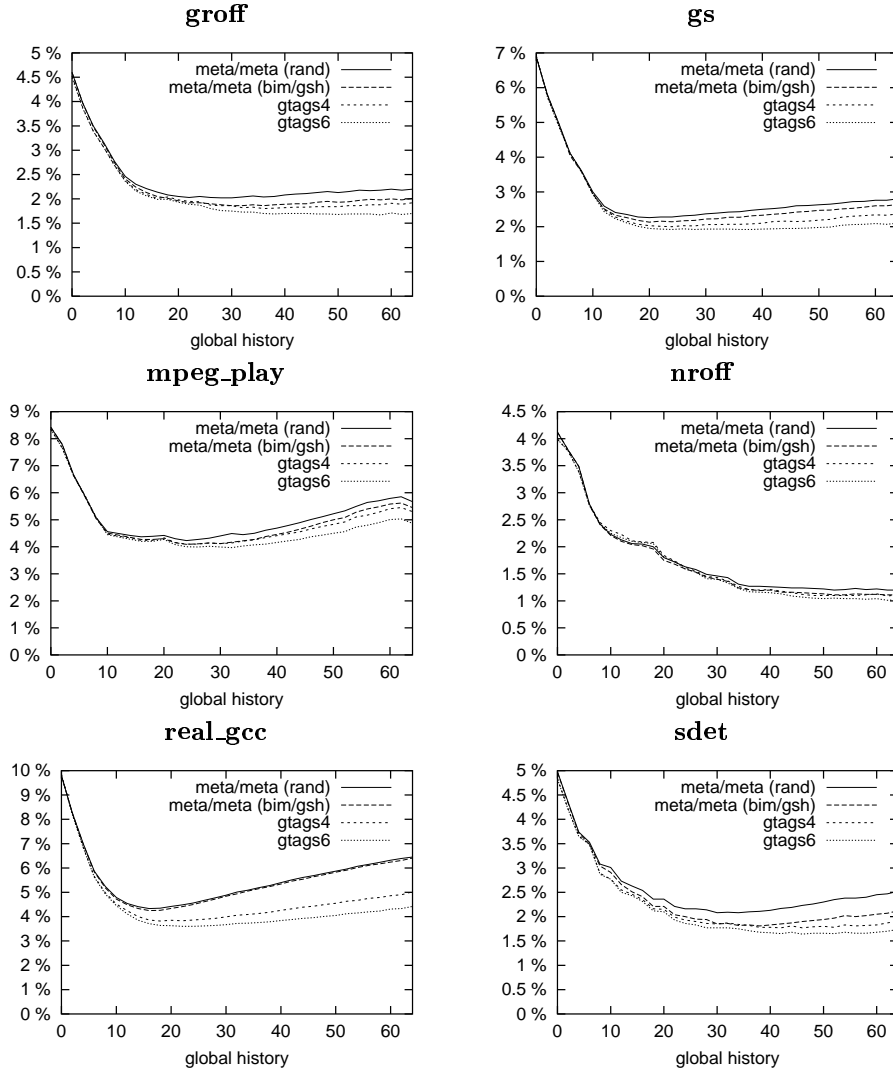
Figure 40: Comparison of gtags4 and meta/meta for a 64 KB budget. Both predictors feature the same quantity of "bimodal" prediction, "gshare" prediction and "selection" information. We show two versions of meta/meta: one initializes the meta-predictors randomly ("rand"), the other initializes the meta-predictors so that the outer meta-predictor selects bimodal and the inner meta-predictor selects gshare ("bim/gsh"). We also show the misprediction percentage of a gtags6 (80 KB).

verilog

video_play

go

mcf

twolf

gap

Figure 41: Cf. Figure 40.

groff

gs

mpeg_play

nroff

real_gcc

sdet
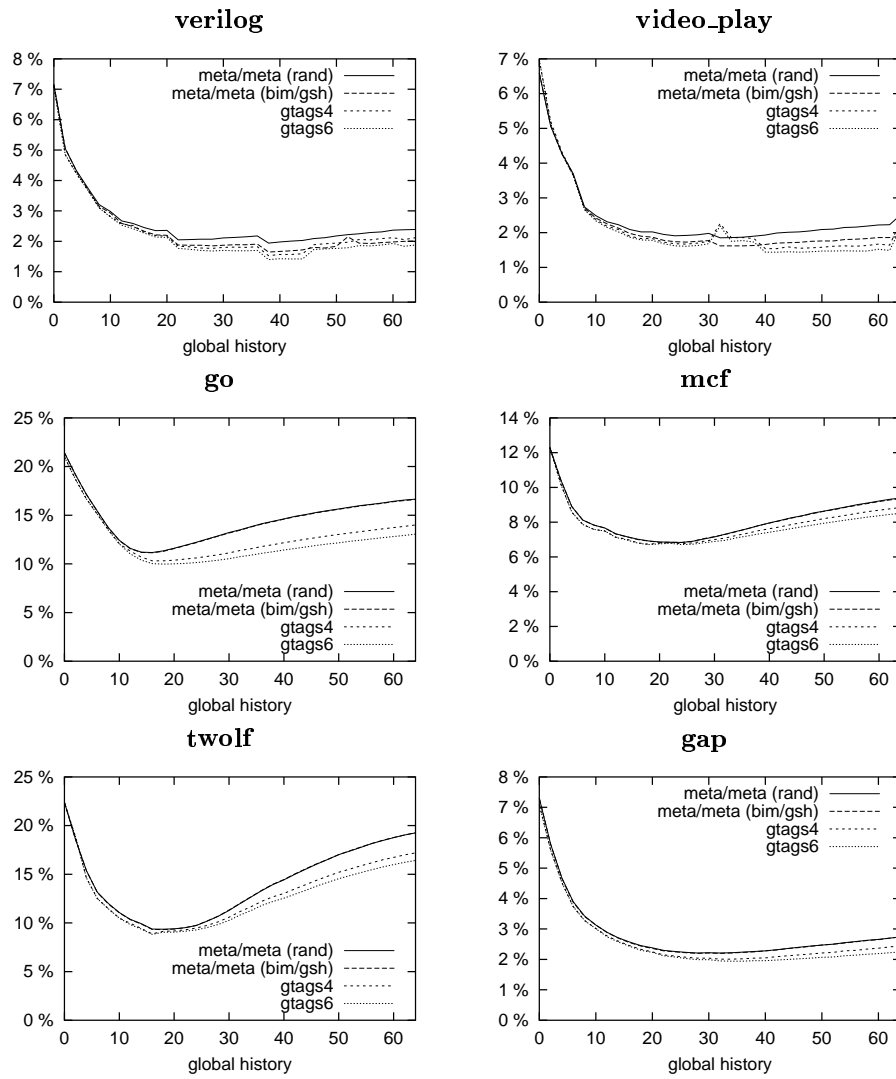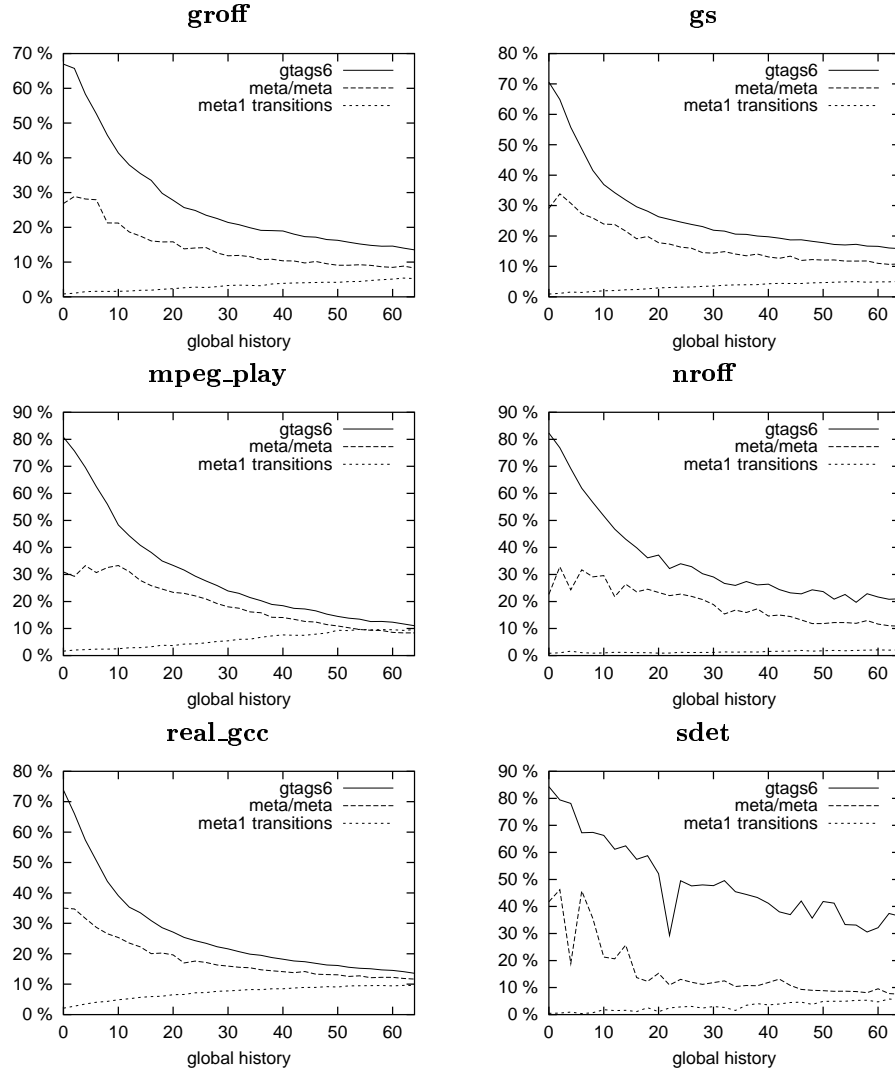
Figure 42: Percentage of dynamic branches using the gshare predictor on a 4-KByte meta/meta and a 5-KByte gtags6. We also show the percentage of T_{meta1} updates in meta/meta that result in a change of the meta-prediction.

verilog

video_play

go

mcf

twolf

gap
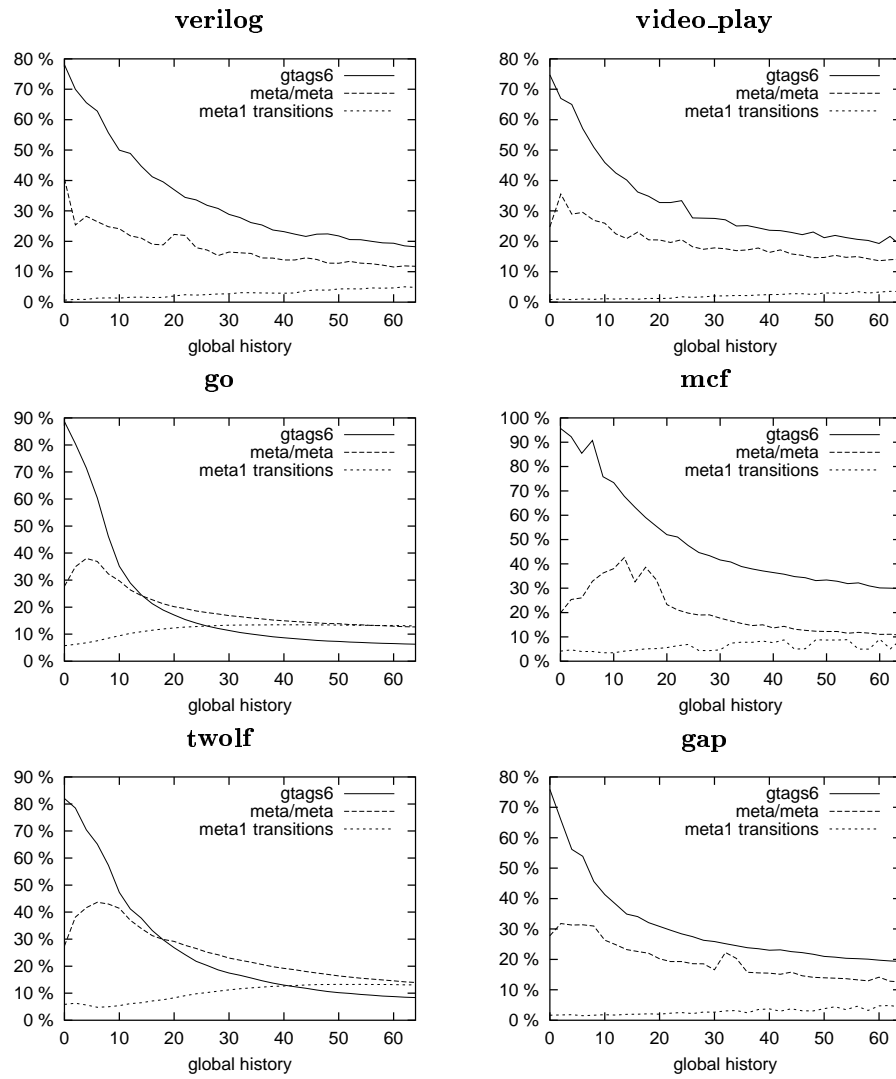
Figure 43: Cf. Figure 42.
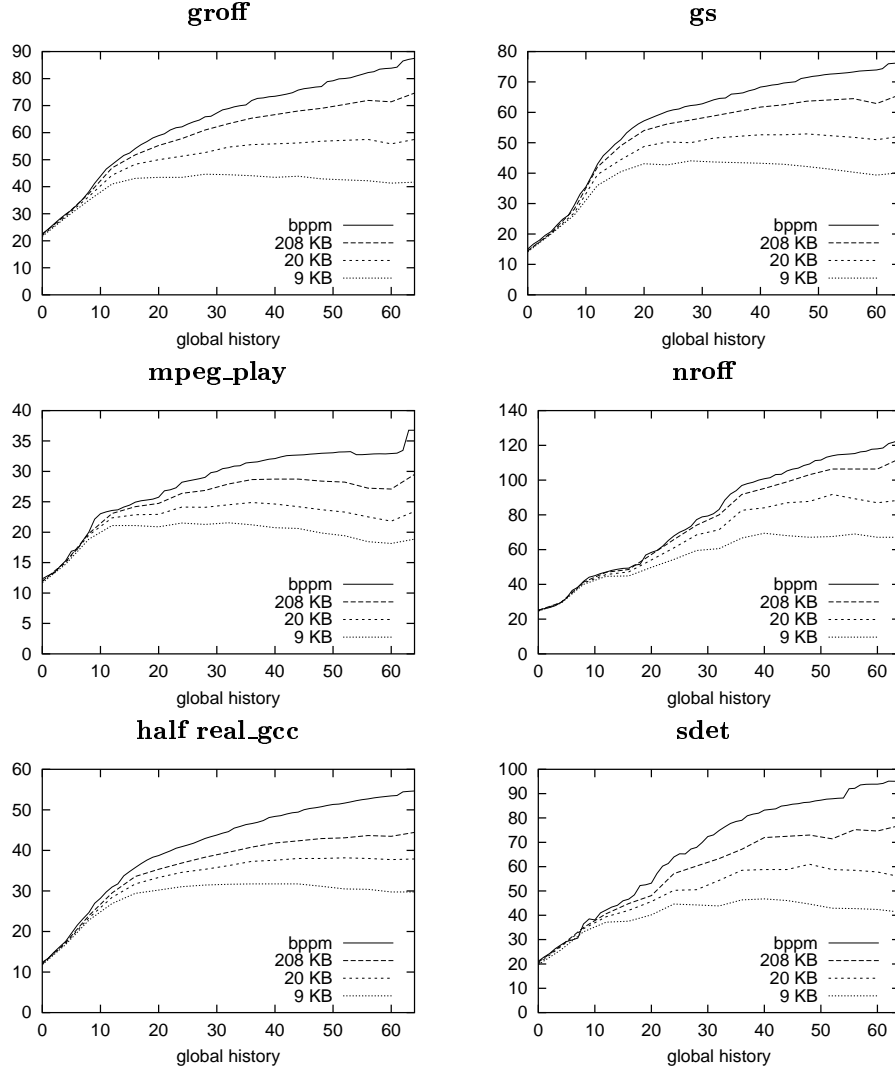
Figure 44: Approximation of BPPM by cascading two match-select. The three tables are $T_1 = gshare1[A, H_h, m]$, $T_2 = gshare1[A, H_{h/4}, m]$ and $T_3 = gshare1[A, 0, m]$, the predictor simulated is $match\text{-}select(T_1, match\text{-}select(T_2, T_3))$. The $gshare3$ function is used for the tags. The y-axis is the misprediction interval (inverse of the misprediction ratio) and the x-axis is the global history h. Three hardware budgets are simulated : 208 KB ($m = 16$, 10-bit tags), 20 KB ($m = 13$, 7-bit tags) and 9 KB ($m = 12$, 6-bit tags).

verilog

video_play

go

mcf

twolf

gap

Figure 45: Cf. Figure 44.

groff

gs

mpeg_play

nroff

half real_gcc

sdet
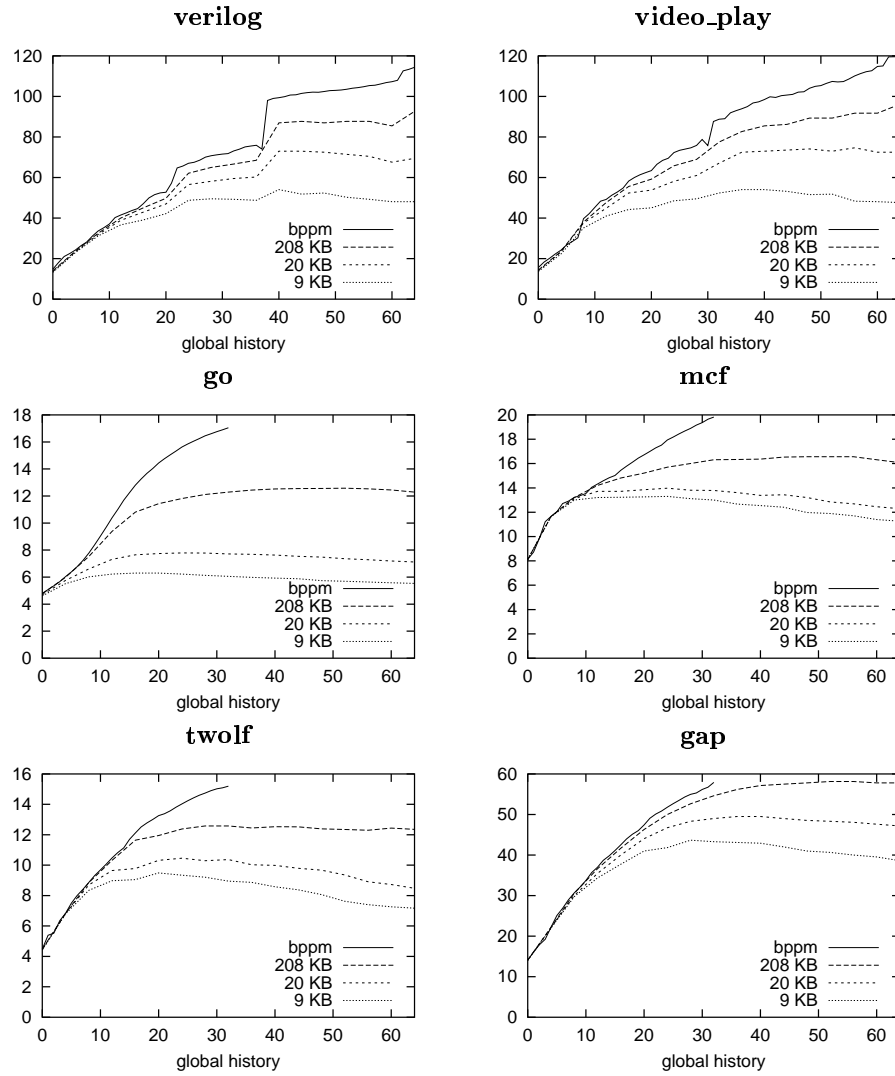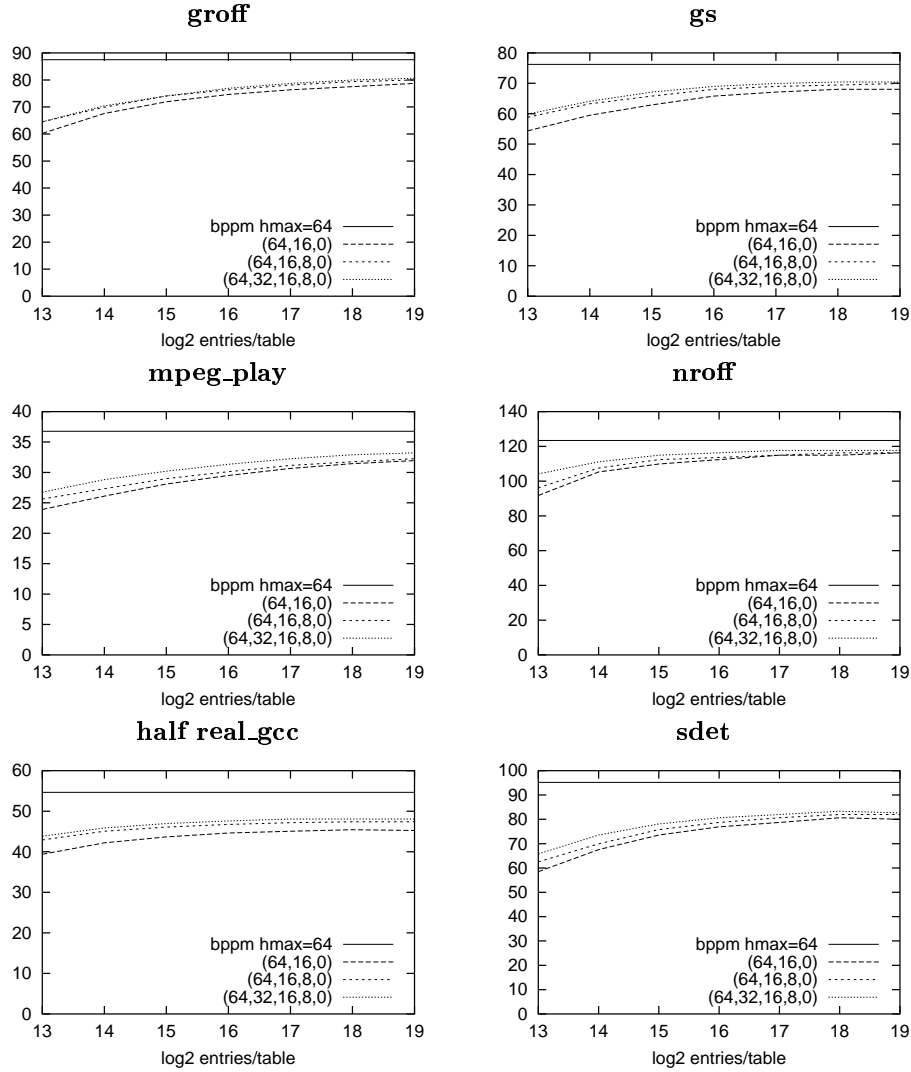
Figure 46: Approximation of BPPM by cascading several match-select (misprediction interval on the y-axis). The maximum global history length is fixed to $h = 64$. On the x-axis, we vary the log_2 of the number of entries per table. The tag width is fixed to 10 bits. Three configurations are displayed : a 3-table configuration using history lengths $(64, 16, 0)$, a 4-table $(64, 16, 8, 0)$ and a 5-table $(64, 32, 16, 8, 0)$.

verilog

video_play

go

mcf

twolf

gap
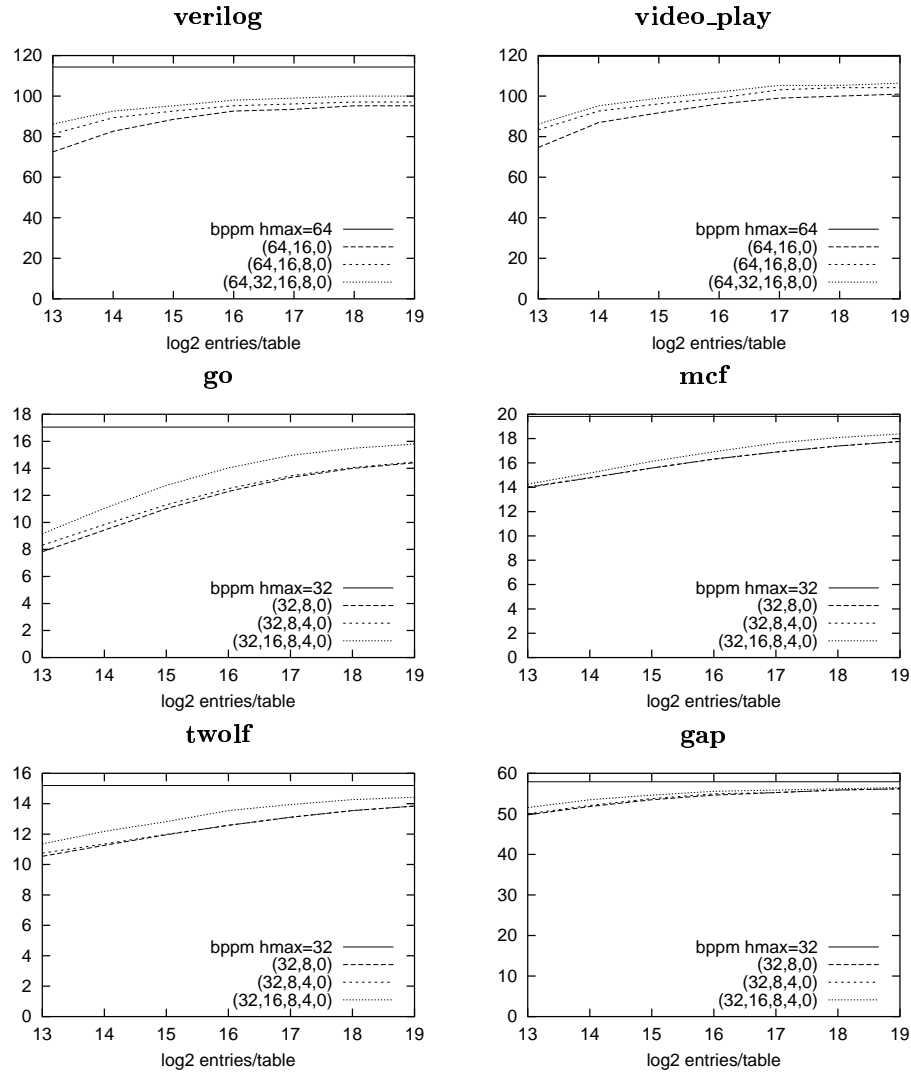
Figure 47: Cf. Figure 46. On the four SPEC benchmarks, the maximum history length is fixed to $h = 32$. Three configurations are displayed : a 3-table configuration using history lengths $(32, 8, 0)$, a 4-table $(32, 8, 4, 0)$ and a 5-table $(32, 16, 8, 4, 0)$.
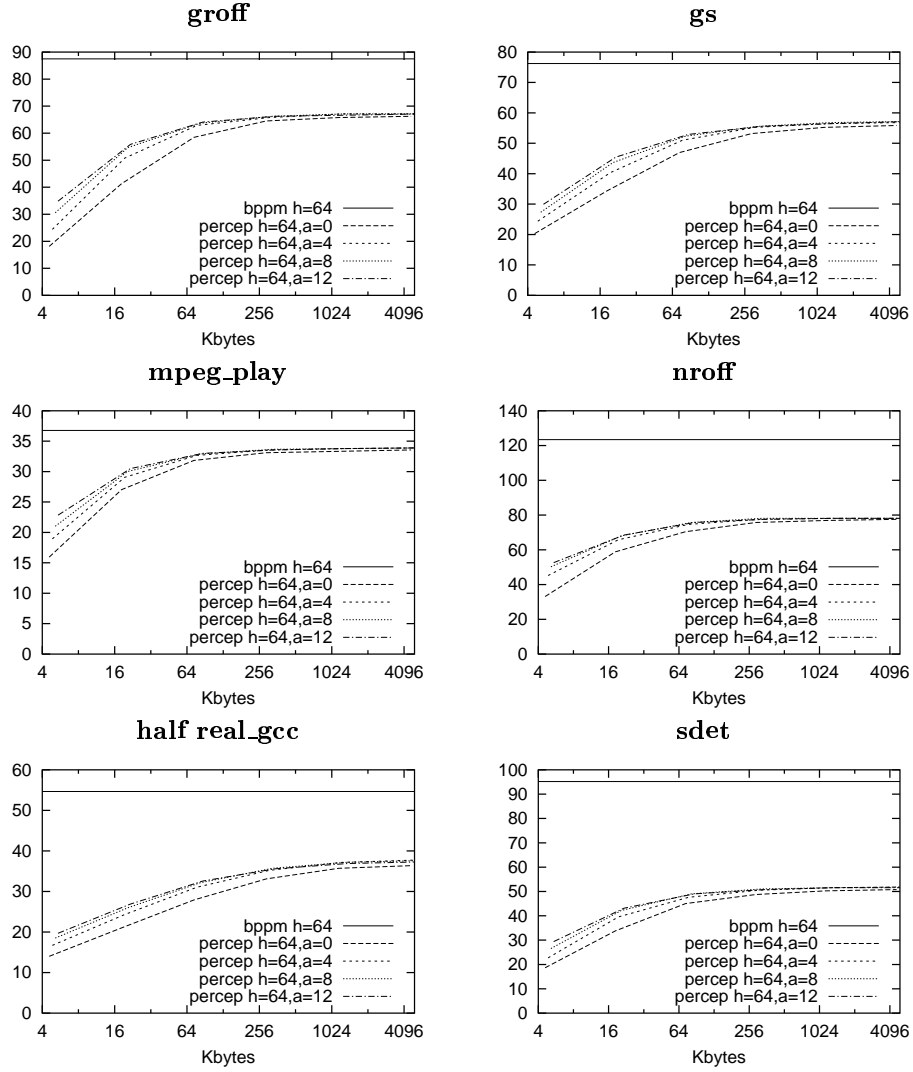
Figure 48: Misprediction interval of a perceptron predictor using a fixed number $h = 64$ of global history bits. We show four perceptron configurations $perceptron[m, H_{64}, A_a]$, with a the number of address bits used as extra inputs. Configuration $a = 0$ is the "normal" perceptron (65 weights per entry). Configuration $a = 12$ uses 12 address bits (77 weights per entry). The number of table entries is varied between 64 ($m = 6$) and 64k ($m = 16$). Perceptron weights are coded on 9 bits. For comparison, we also show the misprediction interval of BPPM with $h = 64$.
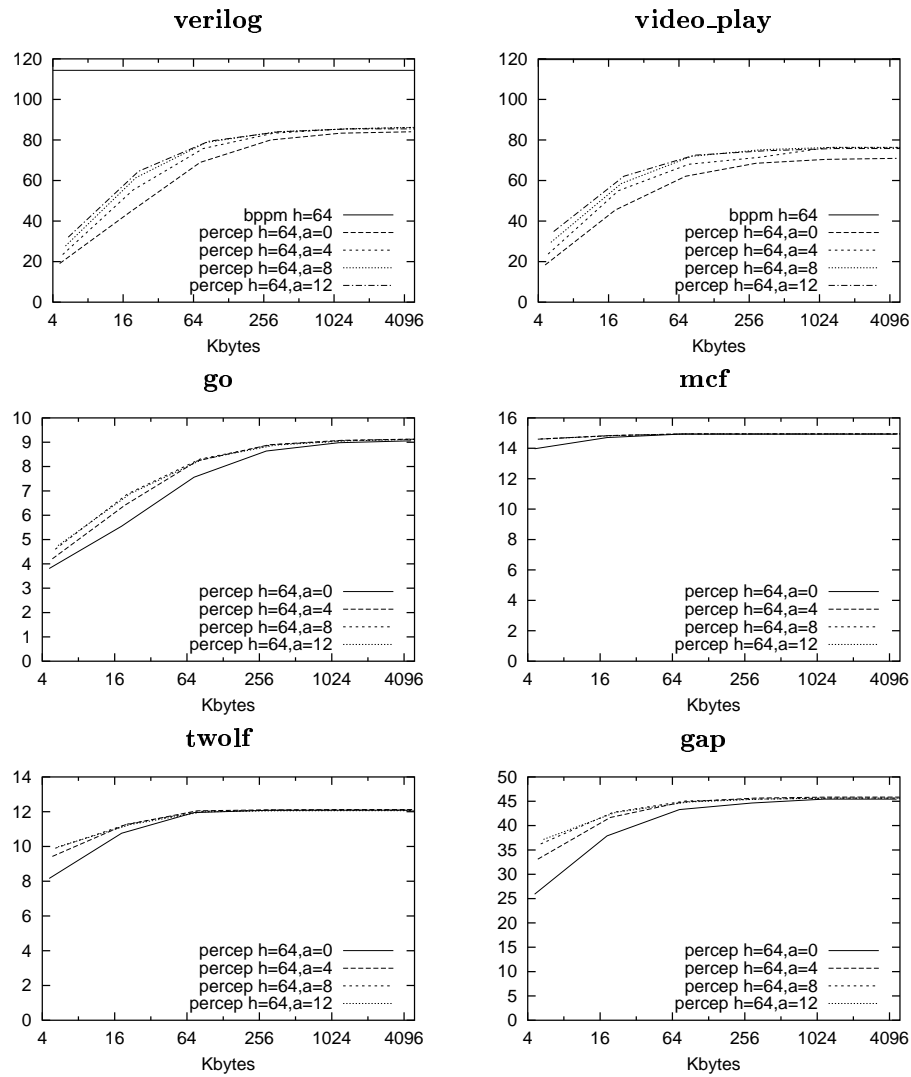
Figure 49: Cf. Figure 48

go

mcf

twolf

gap
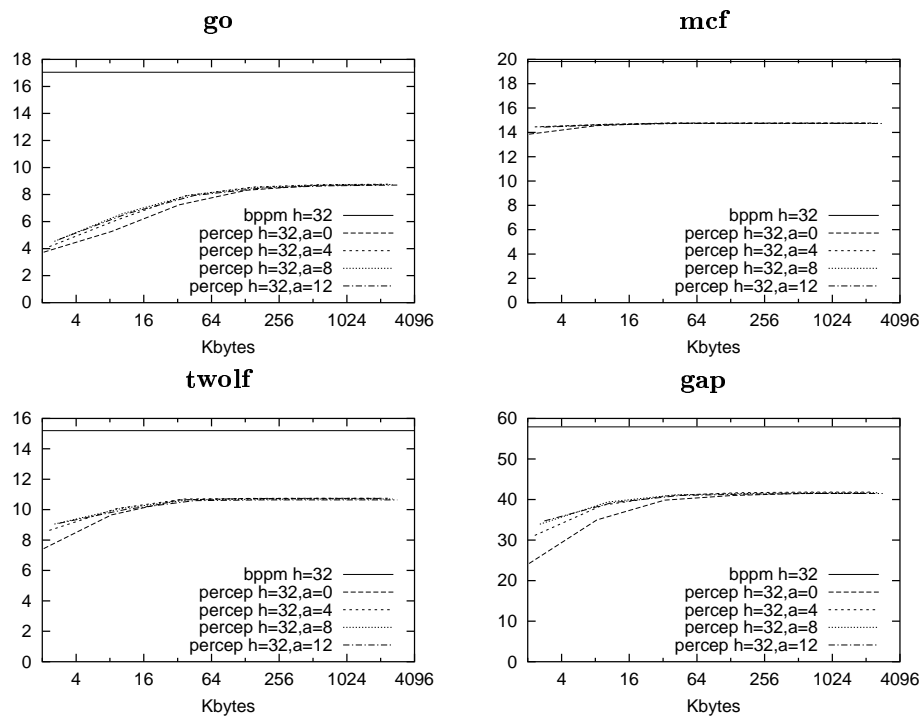
Figure 50: Cf. Figure 49. The global history length *h* fixed to 32 here.

groff

gs

mpeg_play

nroff

half real_gcc

sdet
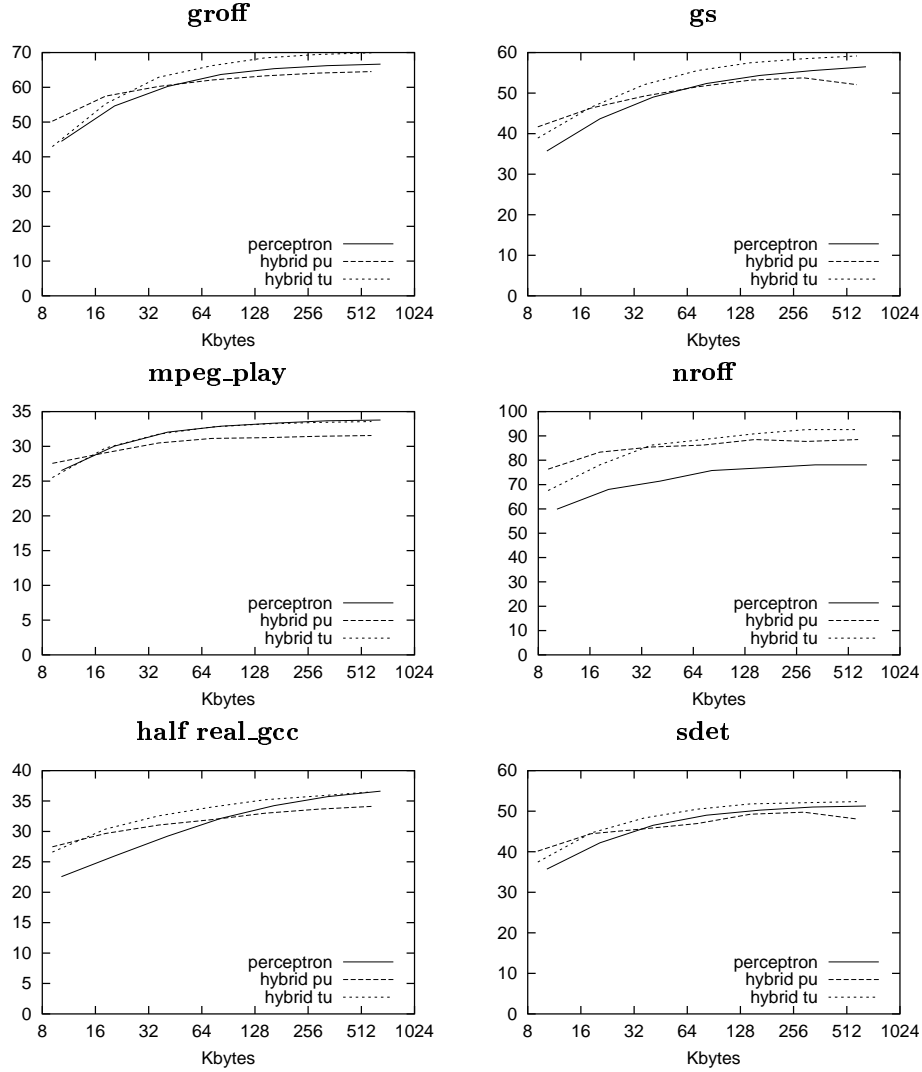
Figure 51: Misprediction interval of a hybrid perceptron. Predictor P_1 is $P_1 = meta\text{-}select(T_1, T_2, T_3)$, with $T_1 = bimodal[A, m]$, $T_2 = gshare1[A, H_{10}, m]$ and $T_3 = gshare2[A, H_{10}, m]$. The hybrid perceptron is $meta\text{-}select(P_1, P_2, T_4)$, with $T_4 = gshare3[A, H_{10}, m]$ and $P_2 = perceptron[m - 6, H_{64}, A_8]$. Parameter m is varied from 12 to 18. Two versions of the hybrid perceptron are shown : partial update ("pu"), and total update ("tu", P_1 and P_2 are always updated). The third predictor is a single $perceptron[m', H_{64}, A_8]$.

verilog

video_play

go

mcf

twolf

gap
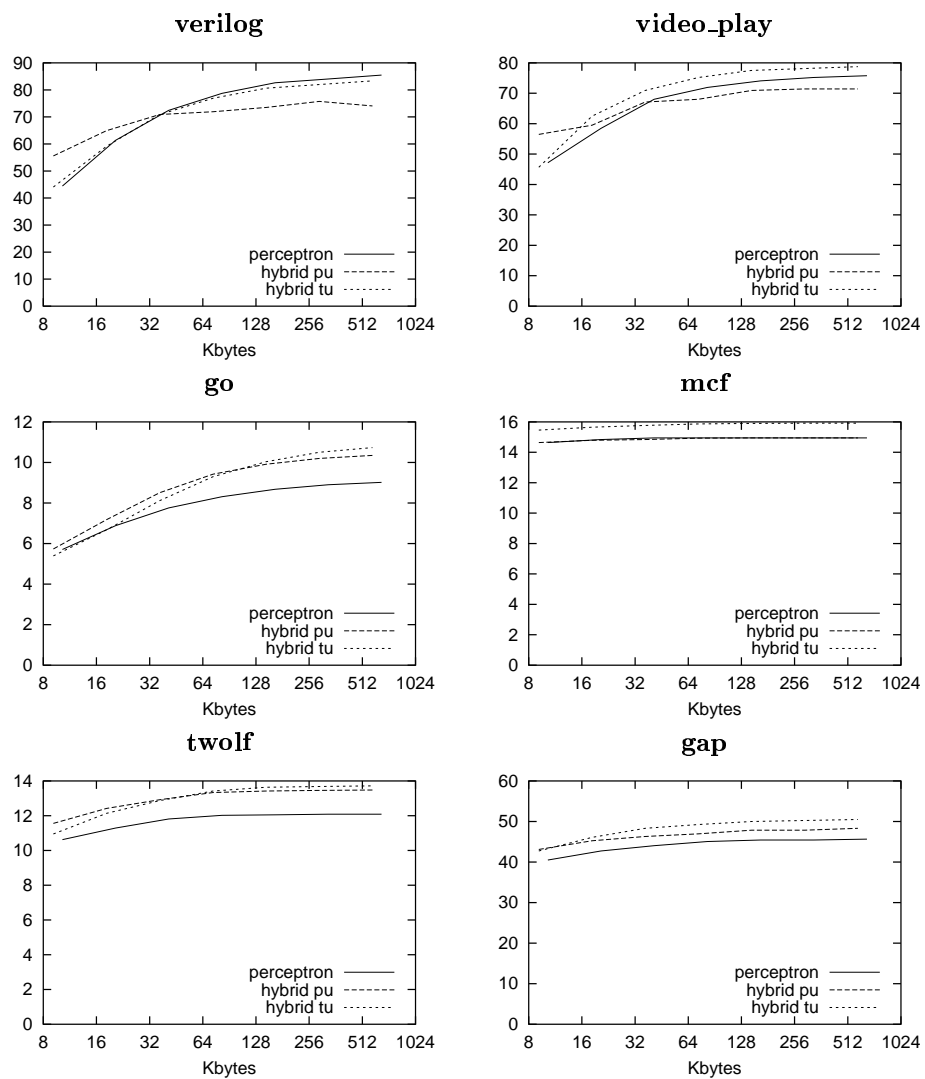
Figure 52: Cf. Figure 51.
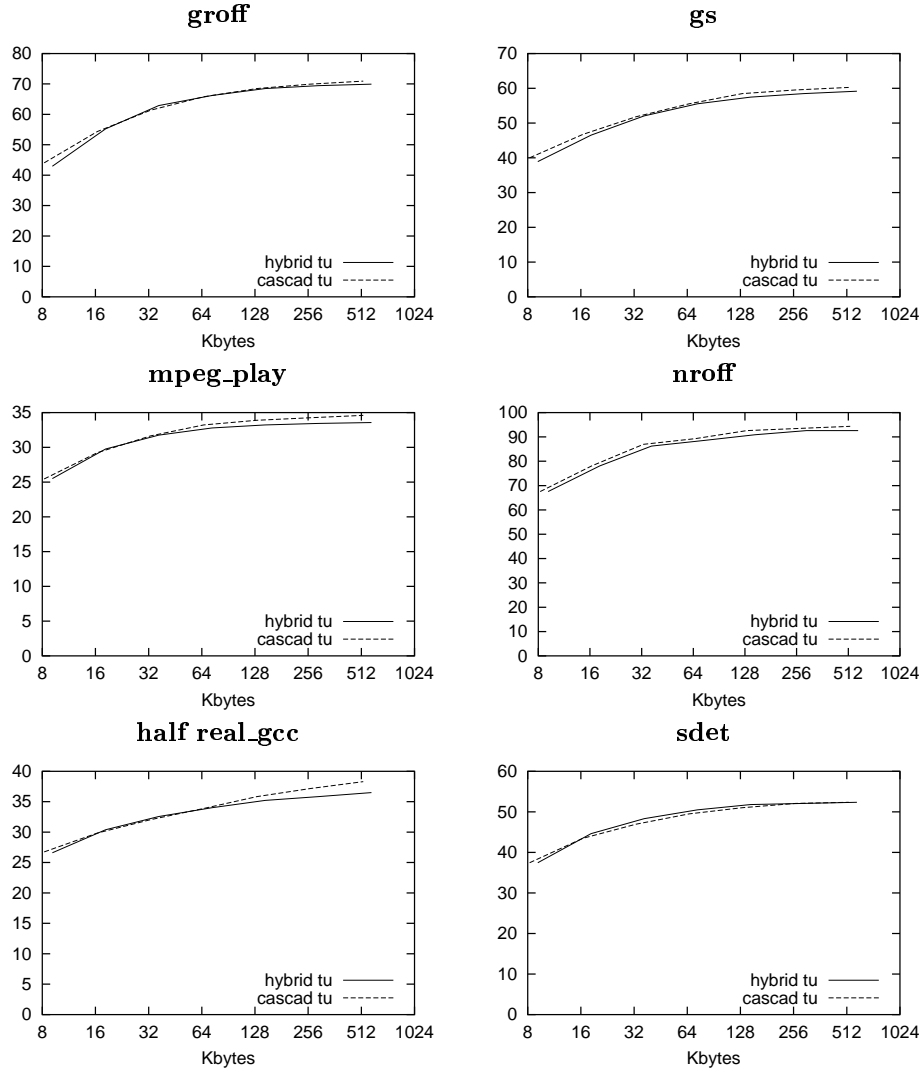
Figure 53: Misprediction interval of a cascaded perceptron, compared with a hybrid perceptron. The hybrid perceptron is the same $meta\text{-}select(P_1, P_2, T_4)$ as on Figure 51, and the cascaded perceptron is $P_2' = perceptron[m - 6, H_{64}, A_8, p1]$, p_1 being the prediction from P_1. Both use a total update (P_1 and P_2/P_2' are always updated). The hardware budget for the cascaded perceptron comprises both P_1 and P_2'.

verilog

video_play

go

mcf

twolf

gap
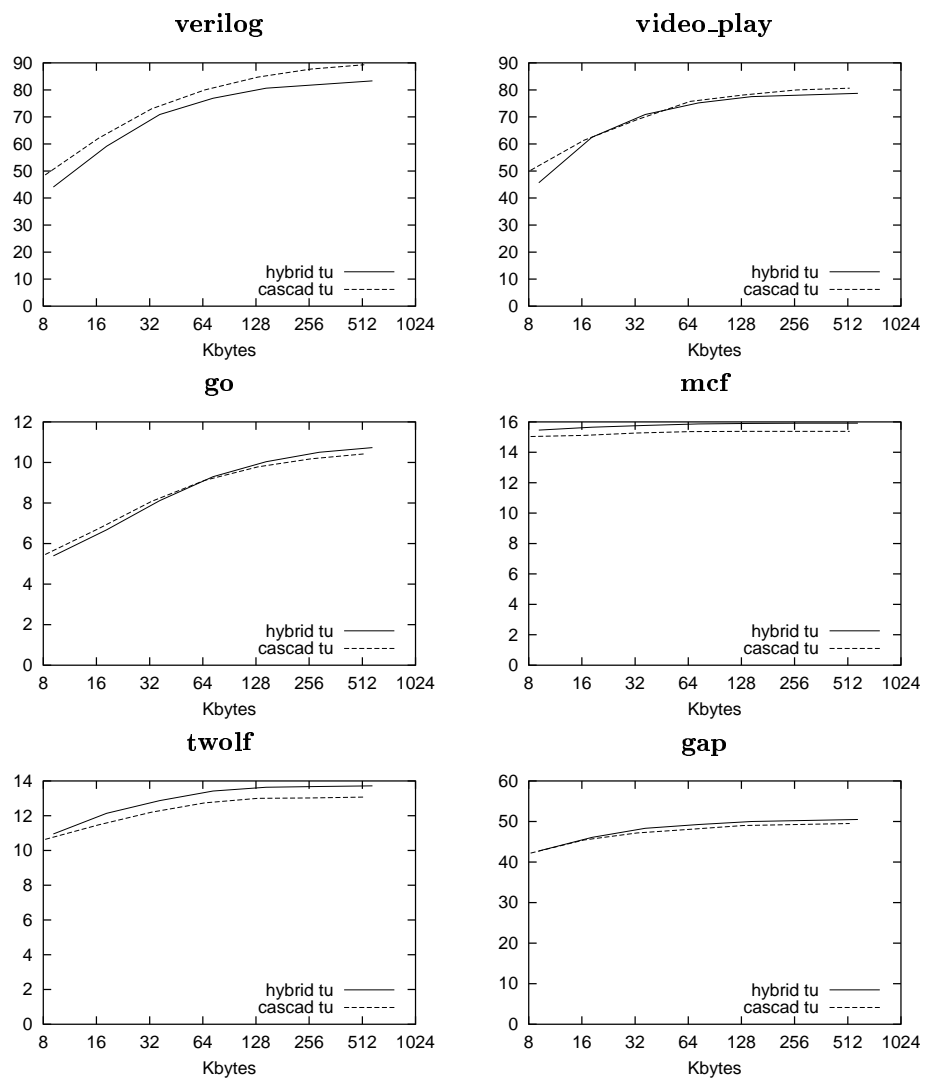
Figure 54: Cf. Figure 53.
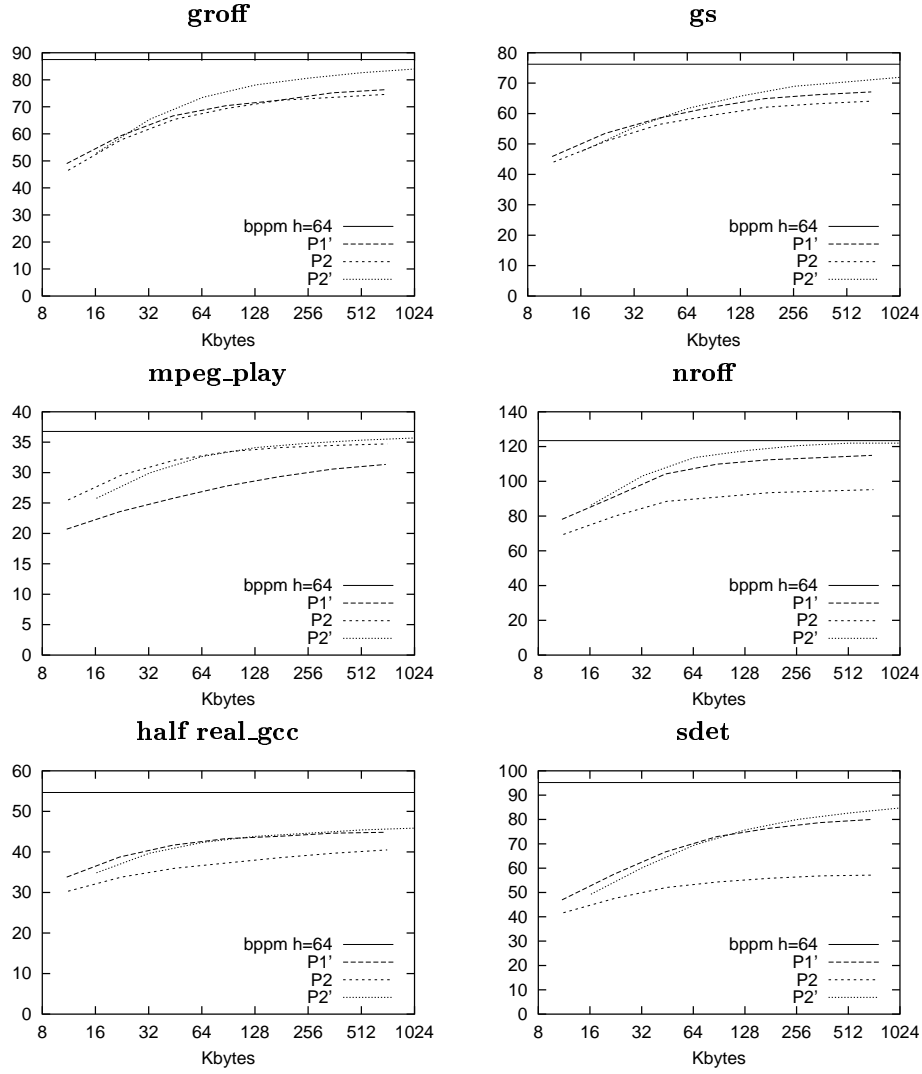
Figure 55: Misprediction interval of a perceptron cascaded with 2-table and 3-table gtags. The gtags are $P_1 = gtags8[16, m] = match\text{-}select(T_2, T_3)$ and $P_1' = gtags8[64, 16, m] = match\text{-}select(T_1, P_1)$, with $T_1 = gshare1[A, H_{64}, m]$, $T_2 = gshare1[A, H_{16}, m]$ and $T_3 = gshare1[A, 0, m]$. The cascaded perceptrons are $P_2 = perceptron[m - 6, H_{64}, A_8, p_1]$ and $P_2' = perceptron[m - 6, H_{64}, A_8, p_1']$. Parameter m is varied from 12 to 18.

verilog

video_play

go

mcf

twolf

gap
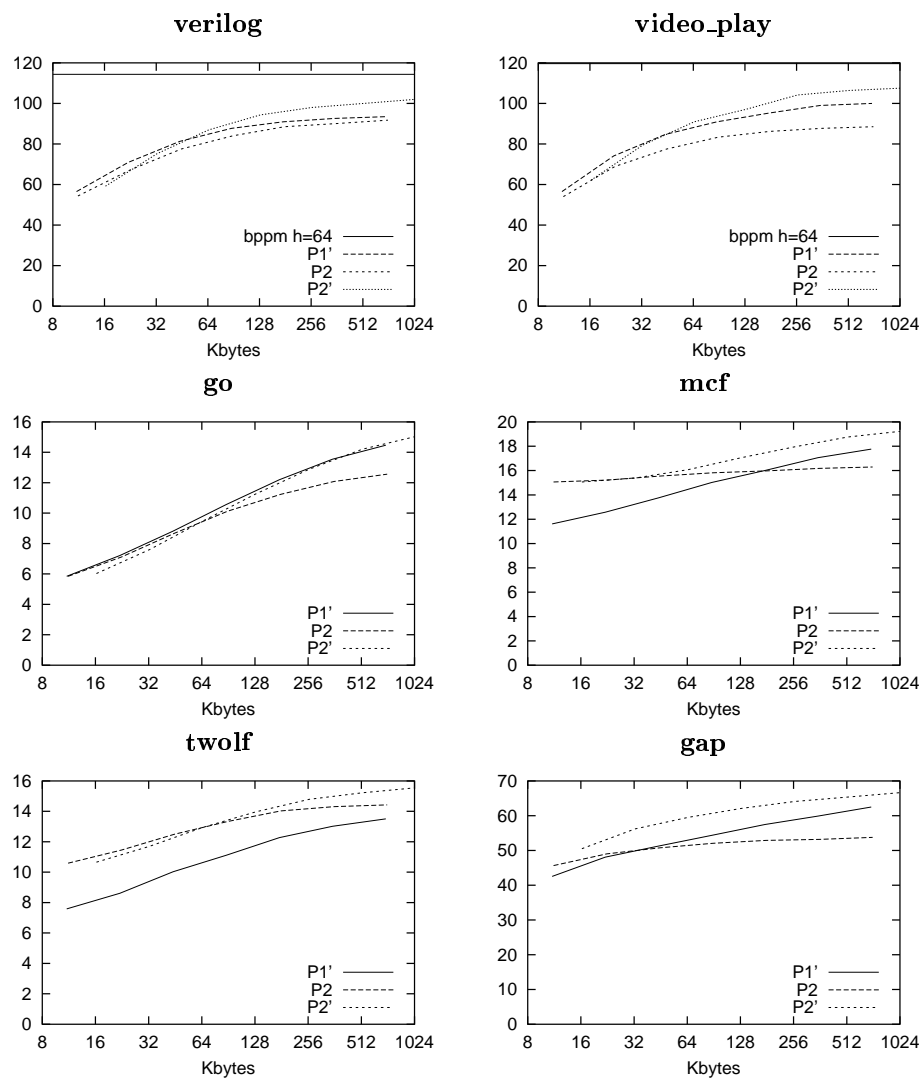
Figure 56: Cf. Figure 55.
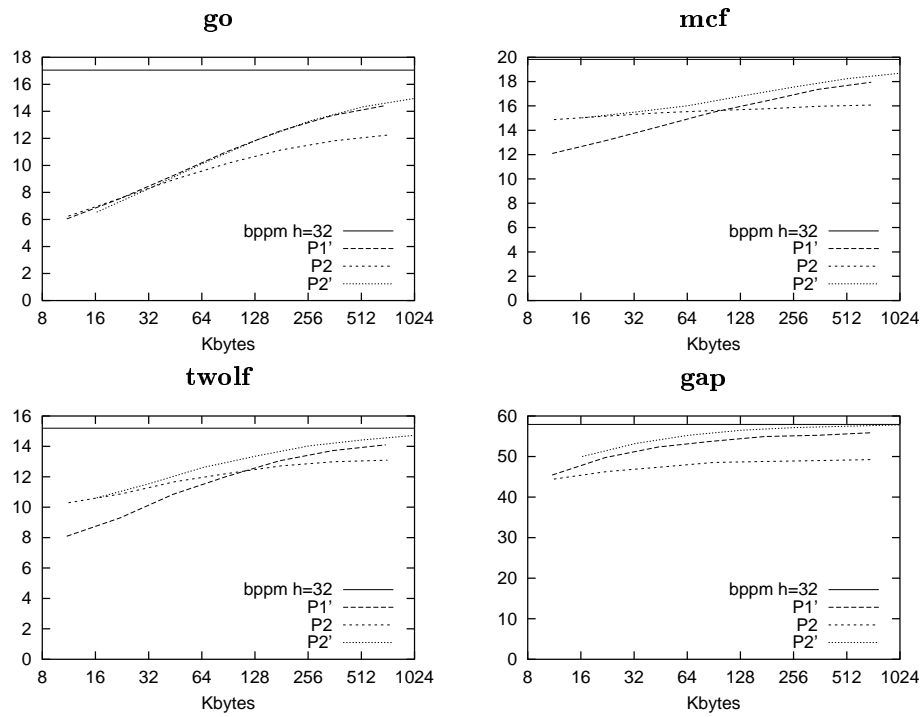
Figure 57: Cf. Figure 56. Here, a global history $h = 32$ is used. The number of perceptron entries have been doubled to keep approximately the same perceptron budget as on Figure 56.