

A Segmented Bloom Filter Algorithm for Efficient Predictors

M. Breternitz¹ Gabriel H. Loh² Bryan Black³ Jeffrey Rupley³ Peter G. Sassone¹
 Wesley Attrot⁴ Youfeng Wu¹

1 - Intel Corporation, 2 - Georgia Tech, College of Computing, 3 - AMD, 4 - UNICAMP

Abstract

Bloom Filters are a technique to reduce the effects of conflicts/interference in hash table-like structures. Conventional hash tables store information in a single location which is susceptible to destructive interference through hash conflicts. A Bloom Filter uses multiple hash functions to store information in several locations, and recombines the information through some voting mechanism. Many microarchitectural predictors use simple single-index hash tables to make binary 0/1 predictions, and Bloom Filters help improve predictor accuracy. However, implementing a true Bloom Filter requires k hash functions, which in turn implies a k -ported hash table, or k sequential accesses. Unfortunately, the area of a hardware table increases quadratically with the port count, increasing costs of area, latency and power consumption. We propose a simple but elegant modification to the Bloom Filter algorithm that uses banking combined with special hash functions that guarantee all hash indexes fall into non-conflicting banks. We evaluate several applications of our Banked Bloom Filter (BBF) prediction in processors: BBF branch prediction, BBF load hit/miss prediction, and BBF last-tag prediction. We show that BBF predictors can provide accurate predictions with substantially less cost than previous techniques.

1. Introduction

Modern processor microarchitectures make use of prediction and speculation in many different ways. Besides the well known branch prediction problem, researchers have studied confidence prediction [9], criticality prediction [7], data-width prediction [12], load hit-miss prediction [24], and others. Many of these prediction problems only require a binary output.

Most microarchitectural binary prediction algorithms make use of a simple table of saturating counters. A hash of an identifier (e.g., instruction address) selects one counter from the table, and the most significant bit of the counter provides the final prediction. Training of the predictor is simple, and consists of either incrementing or decrementing the selected counter based on the true outcome or result. For reasons of circuit latency, chip area, and power consumption, the prediction tables contain a relatively small number of counters (perhaps only a few thousand). This capacity limitation increases the likelihood of different predictions mapping to the same predictor counter (i.e. hash collisions).

Bloom Filters [2] are commonly used in the network and database domains to provide approximately correct answers to set membership queries. The algorithm is easily extended to binary predictions. While Bloom Filters are structurally similar to a table of counters, they differ by employing multiple hash functions to help tolerate conflicts. The Bloom Filter stores each prediction in multiple locations and a combining function (usually a unanimous vote for set membership queries) converts the multiple predictions into the Bloom Filter's final prediction.

The Bloom Filter algorithm can potentially improve the accuracy of microarchitectural binary predictors. However, the latency constraints of most predictor implementations make conventional Bloom Filters impractical. A Bloom Filter with k hash functions would require a table of counters with at least k ports (likely $2k$ ports, k for reading the table and k more for writing), and the area of a multi-ported memory cell increases quadratically with the port count. This paper presents a new version of the Bloom Filter algorithm which is well suited for implementations in hardware and parallel implementations. In particular, banking provides a means of reading multiple entries in parallel without requiring multiple ports, and a special, simple, class of hash functions guarantees that bank conflicts cannot occur. We show the generality of our Banked Bloom Filters with several example applications: branch prediction, load hit-miss prediction [24], and last-tag prediction [5].

Section 2 reviews the basic Bloom Filter algorithm and its applicability to hardware predictors. Section 3 explains our Banked Bloom Filter algorithm and its corresponding implementation. Section 4 describes several applications of Banked Bloom Filters, and Section 5 presents our experimental results. Section 6 provides additional analysis of Banked Bloom Filters, and Section 7 concludes the paper.

2. Bloom Filters

Bloom Filters have traditionally been employed for membership queries [2]. Bloom Filters may occasionally return incorrect answers to queries, but the non-zero error rate enables the Bloom Filter to be implemented in much less state than the information-theoretic lower bounds of a data-structure that guarantees zero errors. Applications include networking [19], databases [3], web caching [6].

For the purposes of set-membership queries, a single-index hash table is at greater risk of returning many false positives. Figure 1(a) shows an element A that belongs to a particular set S . A hash of A provides an index to a par-

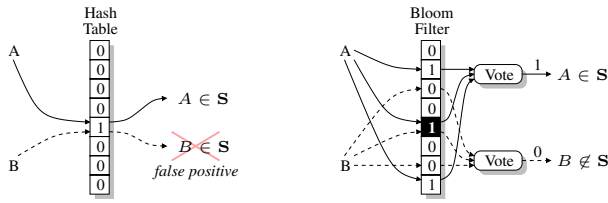


Figure 1. (a)Hash table vs. (b)Bloom Filter

ticular bit in the table, and we set this bit to 1 to indicate membership in S . However, another element B not belonging to S may hash to the same entry as A , which results in the reporting of a false positive.

A Bloom Filter uses multiple hashes for each element, potentially setting several bits in the table for each element that belongs to S . Figure 1(b) shows the same element A hashing into $k = 3$ different locations. In the classical Bloom Filter, an element is considered to be in S when the bits at *all* hashed locations are set. The figure also shows element B and the three entries it hashes to. One of the entries collides with one of A 's entries (shaded); however, there exists at least one other entry that is not set, and so the Bloom Filter correctly classifies B as not belonging to the set. The Bloom Filter is much less likely to report a false positive because hash collisions must occur in each and every one of the k hash functions.

In the context of processor design, Bloom Filters can be used for making 0/1 predictions in the context of processor design. The question “Is branch X taken?” is cast as the membership query of “Does branch X belong to the set of taken branches?” One important difference between traditional Bloom Filters and microarchitectural predictors is that the “sets” in the processor are dynamic. A branch that has been recently taken (belongs to the set) may in the future be not-taken (needs to be removed from the set). A natural modification to the Bloom Filter is to replace each entry with a saturating up/down counter [6]. Predictions can be made based on majority or unanimous voting.

Bloom Filters provide a memory-effective means of tolerating hash conflicts. However, the direct implementation of a Bloom Filter is problematic, because a k -function Bloom Filter requires k lookups from the table. Timing constraints usually make it undesirable, if not infeasible, to perform k serial lookups of the table. Conducting k lookups in parallel requires each SRAM cell in the table to be equipped with at least k read ports, and updating all k counters in parallel adds another k write ports. The size of the SRAM cell increases quadratically with the port count (k wordlines in one dimension, and $2 \cdot k$ bitlines in the other). For the same total storage capacity (in bits), the multi-ported version requires substantially more area, wire length, power and latency. For most hardware applications, Bloom Filters requiring heavy multi-porting would be impractical.

The prior work that most resembles the classic Bloom Filter is the gskewed branch predictor [16]. The gskewed predictor uses k independent tables of counters, each hashed with a different function, and a majority vote deter-

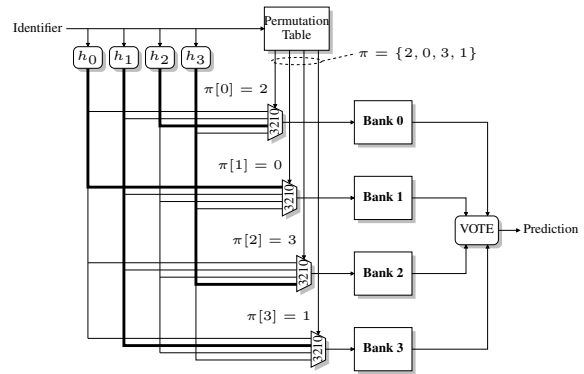


Figure 2. Banked Bloom filter predictor.

mines the final prediction. A disadvantage of the gskewed approach is that for a total capacity of n counters, each of the k individual tables must deal with all branches mapping to only n/k counters. If one table experiences an uneven distribution of usage (high interference in some counters while other counters remain untouched), there is no opportunity for the other tables to make use of the underutilized resources. Sanchez et al [21] compares the performance of parallel banked implementations. Peir et al. also proposed using a Bloom Filter to provide early detection of cache misses [17]. This approach is more similar to gskewed where each hash is constrained to a single sub-table which may experience over/under utilization of entries due to the non-uniform distribution of a single hash function.

3. Banked Bloom Filters

The central difficulty of implementing conventional Bloom Filters is providing the high read/write bandwidth required for the k hash functions. Other hardware tables such as caches frequently use banking as a means of providing bandwidth without adding more ports. A total of B banks with p ports each can provide a maximum of $p \cdot B$ simultaneous accesses *so long as no more than p accesses target the same bank*. For caches, bank conflicts result in stalling until a port is available. However, many hardware predictors must be accessed every cycle, and any delay can potentially back-up the entire pipeline. As a result, banking would be ineffective for implementing conventional Bloom Filters if bank conflicts occur.

We implement Bloom Filters with a banked table to provide bandwidth, and simply *guarantee that conflicts cannot occur* by construction. Conceptually, for a table with B banks (assuming interleaving based on the upper bits), we require that our hash functions h_1, h_2, \dots, h_k observe the following property:

$$\forall x, \forall i, j \in \{1..k, i \neq j\} : \lfloor h_i(x)/B \rfloor \neq \lfloor h_j(x)/B \rfloor$$

That is, for any two hash functions, the resulting indexes *always* reside in different banks. Mathematically deriving such a set of hash functions is not trivial, and the resulting functions may not have practical implementations.

We guarantee conflict-free hashes through the use of a hard-wired *permutation table*. Figure 2 shows a table of

counters organized as four banks and a Bloom Filter-styled access with $k = 4$ hash functions. We use the lookup identifier (e.g., load's PC for hit/miss prediction) to select a bank permutation π from the permutation table. For each bank i , $\pi[i]$ indicates the hash function that should be used. For example, Figure 2 shows $\pi = \{2, 0, 3, 1\}$, and so bank 0 uses h_2 since $\pi[0] = 2$.

Our *Banked Bloom Filters* (BBF) maintain the salient properties of classical Bloom Filters. Each hash function ranges over the entire set of n counters, although any specific lookup identifier maps to only a single bank for that hash function. The even distribution of accesses across the entire table provides for the same level of interference tolerance afforded by a regular Bloom Filter. The hash mapping ensures that bank conflicts cannot occur, and frees the designer to choose *any* hash functions for h_1, h_2, \dots, h_k .

The closest similar approach is using k smaller Bloom filters in parallel [21], each with a table of size m/k where m is the size of the hash table. While asymptotically equivalent, that approach suffers from a higher rate of false positives in situations of higher table loading. This is understood intuitively as follows: False positives are more likely as the table gets full. To increase the likelihood of false positive in the parallel Bloom filters, one needs to fill a good fraction of each of the smaller tables. Since each of the smaller Bloom filters has a lesser range in which to map values, after N insertions it is very likely that each smaller table is highly loaded.

The BBF's overall circuit latency is of the same order as a gskewed predictor. The permutation-table lookup occurs in parallel with the hash function computations. The permutation table itself can be very fast because its fixed contents can be implemented in a hard-wired ROM instead of a conventional SRAM array. For a sufficiently small table, one can implement the permutation table directly in combinatorial logic. The BBF adds the latency of a k -to-1 multiplexer to select a hash function.

4. Applications of Banked Bloom Filters

In this section, we discuss the application of Banked Bloom Filters in several different contexts in microprocessor design, as well as web caching. We describe the application to branch direction prediction, as that is the most common predictor in modern processors. To demonstrate the generality of the BBF technique, we also evaluate BBFs for two very different applications: load data-cache hit/miss prediction and last-tag prediction.

4.1. Branch Prediction

Dynamic branch prediction algorithms are a very well-studied field in computer architecture. Some form of correlation-based prediction algorithm has been implemented in almost all modern high-performance processors. These predictors are typically implemented as a table of counters similar to that described in Figure 1(a). One popular example of such a predictor is the gshare algorithm [14]. While the gshare predictor is not the most accurate algorithm in the literature, we consider gshare in this paper because it is a practical algorithm that is used in many real processor implementations.

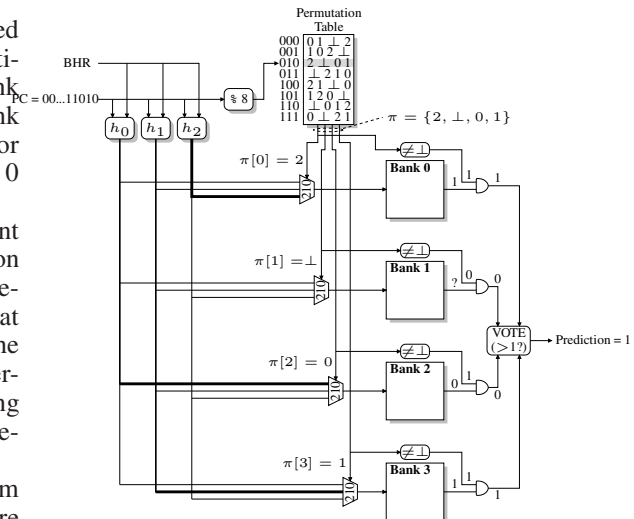


Figure 3. BBF gshare predictor for two different inputs.

The Banked Bloom Filter described above (Figure 2) shows the prediction combining function as a generic “vote” block. We found that a majority vote (as opposed to a unanimous vote) performs better for BBF branch predictors. When using the majority vote, the use of an even number of hash functions necessitates some form of a tie-breaking policy. The simple solution is to implement an odd number of hash functions, and thus an odd number of banks. The majority vote also allows us the use of partial update policies. Here adapt the partial update policy from the gskewed predictor such that we only update correctly predicting banks when the overall prediction was correct, and update all banks on a misprediction.

We have so far made the assumption that the number of banks B in the BBF predictor is equal to the number of hash functions k . However, we can use more banks than hash functions ($k < B$) to decouple the hashing and indexing requirements from the number of banks. For example, consider the BBF gshare illustrated in Figure 3, where $B = 4$ and $k = 3$. In this situation, a branch (plus global branch history) will always hash to three unique banks, which leaves one bank unused. This is not the same as wasting an entire bank, because while branch X may not use bank 3, branch Y might instead ignore bank 1, and Z does not use bank 2, and so on. Across all branches, all banks will statistically be equally utilized.

To accommodate fewer hash tables than banks, we modify the permutation table to include ‘ \perp ’ which denotes that the bank should not be used for the current prediction. Since there are only $k = 3$ hash functions, each bank’s index multiplexer selects from only one of k possible inputs. Out of B banks, only k will receive valid indexes, and the remaining will be assigned \perp ’s. In parallel with the bank lookups, a simple logic gate checks to see whether the corresponding bank has been assigned \perp . If the majority function is imple-

mented with an adder or counting gate, a simple AND gate is sufficient to disable the corresponding bank's output.

To better illustrate the operation of the BBF gshare predictor, Figure 3(a) shows the indexing process for a branch with an address ending in $\dots 010_2$. We use an eight-entry permutation table. The three lowest bits of the branch address select entry 010 of the permutation table, yielding $\pi = \{2, \perp, 0, 1\}$. The three hash functions yield three different indexes. In particular, we use the same hash functions employed by the enhanced version of the gskewed predictor [16]. The function h_0 is a PC-only hash, and h_1 and h_2 combine PC and branch history. The bold lines illustrate which hash functions provide the indexes to the different banks. Note that since $\pi[1] = \perp$, none of the three hashes ended up getting mapped to Bank 1. Each bank provides the value of the indexed saturating counter, and the most significant bit of the counter represents the bank's prediction. The voting logic combines the bank predictions with a 4×1 -bit adder, and produces a final prediction of 1 for a taken branch. Figure 3(b) shows another example using a different branch address and branch history. Notice that in this example, the hash functions end up mapping to different banks and use a different set of banks.

The contents of the permutation table help spread accesses evenly across all B banks. In the examples of Figure 3, the different permutations were chosen such that each table has an equal probability of having a particular hash function mapped to it. For $B = 4$ and $k = 3$, we make sure each symbol in $\{0, 1, 2, \perp\}$ appears twice per column. In general, for any $k < B$, each row of the permutation table is a permutation of:

$$\{0, 1, 2, \dots, k-1, \overbrace{\perp, \perp, \dots, \perp}^{B-k \text{ times}}\}$$

A larger number of rows in the permutation table allows for a more even distribution of branches across the banks of the predictor (we explore the impact of changing the permutations in Section 6). We use the permutation table shown in Figure 3 for our experimental results in the next section.

In principle, there are no constraints on the number of banks and the number of hash functions so long as $k \leq B$. However in practice, too large a value for either k or B introduces a lot more complexity in routing the hash functions to the B banks and then selecting k out of B outputs and performing the final vote.

4.2. Load Hit/Miss Prediction

Besides branch direction prediction, many computer architecture, studies have considered other uses of binary predictors in processor design. Some high-frequency microarchitectures use speculative scheduling to meet timing constraints. When a load instruction issues, its dependents may be scheduled on the assumption that the load will hit in the L1 data cache and have a corresponding execution latency. If the load misses in the cache, then the incorrectly scheduled dependent instructions must be *replayed*. A flush may cause rescheduling of all instructions in the scheduler pipeline or perhaps the dependent instructions in the load's forward slice must be replayed.

One technique to reduce the frequency of replays is load hit/miss prediction [24]. Based on the past history of a load's hit/miss patterns, one can predict whether the load will miss again in the future. If a load is predicted to suffer an L1 data cache miss, then the scheduling logic may choose to schedule the load's dependents assuming the L2 latency as opposed to the L1. In this situation, a replay can be avoided which allows other independent instructions to issue. Predicting a cache miss when the load actually hits in the L1 delays the scheduling of a load's dependents, which may decrease performance. Yoaz et al. proposed a hybrid load hit/miss predictor, where one component is a gshare-like predictor [24]. In this work, we consider applying the BBF technique to this gshare-like hit/miss predictor. Our goal is not to develop the best possible load hit/miss predictor, rather we are only demonstrating that Banked Bloom Filters can help whenever a gshare-like predictor is used.

4.3. Tag Elimination

As another example of the applicability of BBFs, we consider Ernst and Austin's idea of *Last-Tag Prediction* [5]. The reservation stations of a dynamically scheduled processor track the readiness of all of an instruction's operands. The insight of the last-tag prediction work is that for the scheduling problem, only the last arriving operand actually signifies the readiness of the instruction. By predicting which of an instruction's two operands will be the last to arrive, the scheduler can be simplified by implementing only a single set of CAMs for matching against the last-predicted to arrive operand.

In Ernst and Austin's last-tag prediction work, they made use of a gshare-style last-tag predictor. For each instruction, a hash of the instruction's PC and the current branch history indexes into a table of saturating counters. The most significant bit of the counter indicates whether the left (0) operand or the right operand (1) will be the last to arrive. On a misprediction, the processor suffers from an instruction scheduling replay, and therefore the last-tag prediction accuracy affects both performance and power.

Banked Bloom Filters can be applied to Last-Tag Prediction. Since the last-tag predictor is structurally identical to the gshare branch predictor, the hardware for a BBF last-tag predictor is basically the same as the BBF gshare.

4.4. Multiprocessors and Web Caching

Another example of the applicability of BBFs is in web caching [6]. In web servers it is often desirable to consult a memory-resident acceleration structure to check the presence of web pages before initiating an expensive I/O operation. Multi-gigabyte Bloom filters are envisioned in this situation. Many data centers split the processing task to multiple processors, so an efficient parallel implementation is desirable [4].

A conventional software implementation would perform k sequential accesses, incurring in extra latency and precluding the benefits of multiprocessor implementation. An alternative multiprocessor implementation could attempt issuing k concurrent accesses from k processors. However, this might suffer from false-sharing conflicts whenever two

processors map to the same cache line. A software implementation of our proposal solution avoids such conflicts by ensuring that only one processor maps to a given cache line: each ‘bank’ is defined as the low-order bits designating the address of the target cache line.

5. Results

In this section, we present our experimental results quantifying the benefits of using Banked Bloom Filters for microarchitectural binary predictors. We first explain our simulation methodology, and then discuss the results for our example BBF applications.

5.1. Methodology

For all three predictor applications, we used the SimpleScalar toolset for the Alpha ISA. In particular, we used the in-order simulator `sim-bpred` for the branch prediction studies, the in-order cache simulator `sim-cache` for the load hit/miss predictor evaluation, and the cycle-level timing simulator MASE [10] for the last-tag prediction experiments. We used applications from SpecCPU (integer and floating point), MediaBench [11], graphics programs from the SimpleScalar website, and others totaling to 100 traces. We used SimPoint to select representative samples of 100 million instructions [18].

For the branch prediction studies, we re-simulated the predictors using a framework similar to that provided by the Championship Branch Predictor competition [1]. We drive the framework with these Intel® traces containing x86 instructions with lengths varying from 30 million to 100 million μ ops from SpecCPU, multimedia, workstation, server, digital home, games, office and productivity applications, totaling to 575 traces. We include these simulation results to increase confidence in the general applicability of Banked Bloom Filters.

5.2. BBF Branch Prediction

Figure 4 shows the average misprediction rates (in mispredictions per thousand instructions or MPKI) of gshare, gskewed and the BBF gshare ($B = 4, k = 3$) predictors. In particular, Figure 4(a) shows the results averaged across all benchmarks, and Figure 4(b) shows results averaged over SpecInt only. In both cases, these results are from our SimpleScalar-based simulations.

The Banked Bloom Filter technique improves gshare’s ability to tolerate interference. In particular, Figure 4(a) shows that a 4KB BBF gshare achieves the same prediction accuracy (actually slightly better) than a 12KB gskewed and a 32KB conventional gshare (these three configurations are marked with Δ ’s in the figure). That is, the 4KB BBF gshare delivers the same performance with only 1/3 and 1/8 the number of counters than gskewed and gshare, respectively. The 8KB BBF gshare performs similarly compared to the 24KB gskewed and 64KB gshare. At the smaller hardware budgets, the difference is not as pronounced, but the BBF gshare still consistently provides better prediction accuracy than the competition. For the SpecInt applications shown in Figure 4(b), the absolute misprediction rates are higher, but the overall trends are similar. Depending on the hardware budget, the BBF gshare performs as well as a conventional

gshare two to four times its size. Except for the smallest size evaluated, a BBF gshare provides the same or better interference tolerance as a 50%-larger gskewed predictor.

We simulated a large variety of traces to demonstrate the robustness of the Banked Bloom Filter technique. We also include results from an Intel® simulator and trace set to provide additional evidence to bolster our claims of BBF’s general applicability. Figure 5 shows the average misprediction rates by application group for 4KB gshare and BBF gshare, and 6KB gskewed. Figure 5(a) shows the results for the academic traces, and Figure 5(b) shows the results for the industry traces (in mispredictions per thousand μ ops or MPK μ). Given the differences in instruction sets (x86 vs. Alpha), applications evaluated, and sampling methodologies, it is not surprising that the absolute misprediction rates vary a bit. However, the general trends are similar: the 4KB BBF gshare provides a substantial reduction in the overall misprediction rate over a 4KB conventional gshare (10.2% overall for the industry traces), and similar accuracy to the gskewed predictor while requiring 1/3 fewer counters.

5.3. BBF Load Hit/Miss Prediction

We evaluated the applicability of Banked Bloom Filters to a gshare-style load hit/miss predictor. For both the conventional and BBF-style predictors, the prediction accuracy when there is a cache hit is 99% or better. Therefore, we will only focus on the prediction of cache misses. Accurate prediction of cache misses is often difficult because cache misses are infrequent and are highly dependent on the dynamic behavior of data access patterns. Figure 6 shows the results for conventional and BBF load hit/miss predictors assuming an 8KB and a 16KB L1 data cache. The difficulty of accurately predicting a load miss increases as the frequency of load misses decreases, which is why the absolute misprediction rates increase for the 16KB cache configurations.¹ While load hit/miss prediction remains difficult to accurately predict, the application of the BBF technique reduces interference by enough such that the conventional gshare predictor needs twice as many counters to match the accuracy of the BBF approach.

5.4. BBF Last-Tag Prediction

To evaluate the impact of Banked Bloom Filters on a last-tag predictor, we simulated a six-wide processor with a 256-entry ROB, 64 RS entries, 64-entry load queue and store queue, a 20-stage pipeline (fetch to branch execute), and 16KB, 512KB and 4MB L1, L2 and L3 caches, respectively. Similar to the original study by Ernst and Austin, we assume a gshare-style last-tag predictor and a single-cycle reschedule (replay) penalty on a last-tag misprediction. The single-cycle replay penalty does not cause much difference in IPC rates, and therefore we only present the misprediction rates. Figure 7 shows the last-tag misprediction rates with and without the BBF approach for a range of predictor sizes, averaged over the 100 academic traces. While the

¹The original work by Yoaz et al. reported a variety of misprediction metrics normalized against the *total* number of loads (whereas we normalize against the number of load misses), which makes their absolute rates appear to be much smaller.

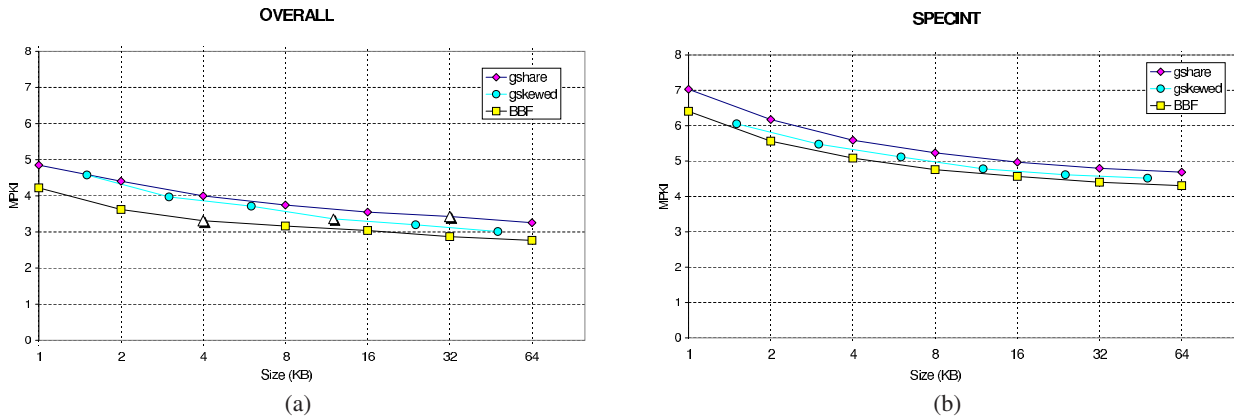


Figure 4. Branch misprediction rates (a) over all of the academic traces and (b) only for SpecInt.

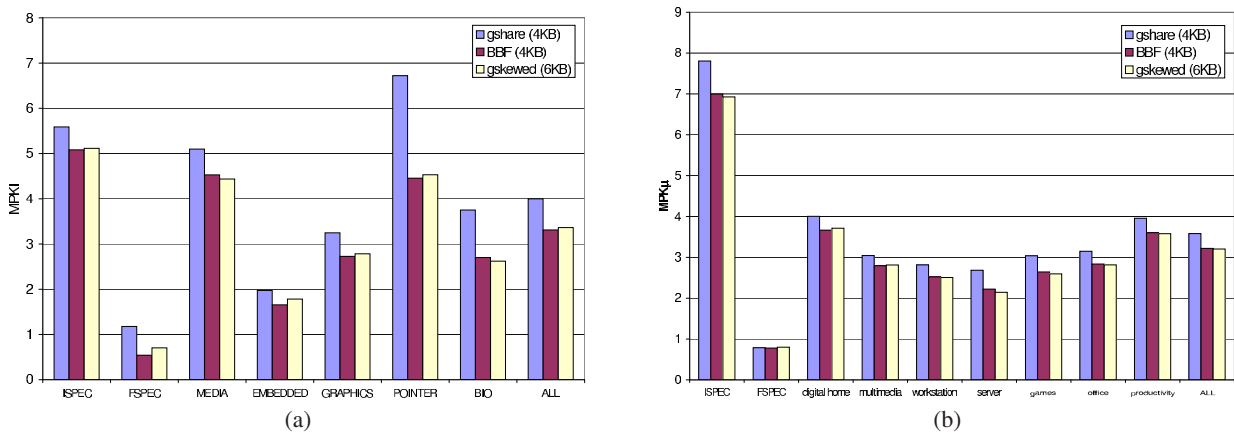


Figure 5. Branch misprediction rates per application group for (a) the academic traces and (b) the industry traces.

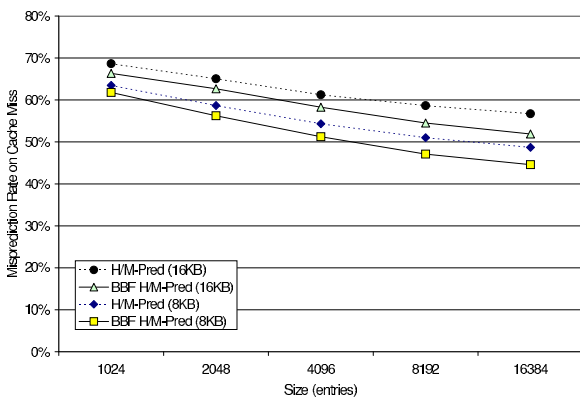


Figure 6. Load hit/miss predictor misprediction rates (percent predicted hit-actual miss).

results are not as dramatic as for the BBF branch predictors, the BBF technique still provides better accuracy than a conventional gshare approach. The interference reduction provides a 6-7% reduction in last-tag mispredictions for 1K- to 4K-entry predictors over the baseline last-tag predictor.

6. Design Analysis

The previous section demonstrated the benefits of Banked Bloom Filters for a variety of hardware binary predictors. Apart from varying the size of the predictors, so far we only considered one possible implementation of BBF predictors. This section explores some of the other design options and discusses their benefits or shortcomings.

6.1. Summation versus Voting

Each entry of the predictors contains a saturating counter, where we only take a vote using the most significant bit. Another approach would be to predict using the summation of the counter values, known as a weighted majority. The idea is that if one counter never/rarely suffers from interference and is stuck in a (say) strongly taken state,

