A Study of Load Hit/Miss Predictors [*]

Nathanaël Prémillieu
ENS Cachan, Brittany extension
University of Rennes 1
nathanael.premilleu@ens-cachan.org

Hans Vandierendonck
Department of Electronics and Information Systems
Ghent University
Hans.Vandierendonck@elis.ugent.be

September 4, 2008

Abstract

Off-chip memory accesses take a long time on modern processors. As such, load instructions that miss in the last-level cache significantly reduce the efficiency of the processor. Various architectural optimizations improve performance and/or reduce energy by predicting these off-chip memory accesses before executing the load instruction.

A hit/miss predictor predicts whether a load will hit or miss in a cache once it executes. Hit/miss predictors have been studied independently by a number of researchers. These researchers borrow ideas from branch prediction and use some techniques specific to hit/miss prediction. It is, however, not clear what the optimal design for a hit/miss predictor is. In this report, we perform a first thorough analysis of hit/miss prediction. We first evaluate various prediction schemes described in the literature. Then we analyze the best predictor from the literature and apply several optimizations to increase its accuracy.

[*]Internship report, Second year, Magistère Informatique et Télécommunications, ENS Cachan and University Rennes 1. Internship conducted in the PARIS team of Ghent University under the guidance of Hans Vandierendonck

Contents

1 Introduction 3

2 Evaluation environment 4

2.1 Benchmarks 4

2.2 Simulations 4

2.3 Exploitation of the results 4

3 Analysis of the benchmarks 5

4 Load Hit/Miss predictors 10

4.1 Simple predictors 10

4.2 History-based predictor 11

5 Improvements and results for the Per-address predictors 14

5.1 Improvements 14

5.1.1 Size of the first level table 14

5.1.2 Size of the history 15

5.1.3 Tags 17

5.2 Other experiments 18

5.2.1 Number of bits of the counters 18

5.2.2 Filtering of the entries 18

5.2.3 Aliasing in the second level table 19

6 Conclusion 19

1 Introduction

As a corollary of Moore's law, every 18 months performance of microprocessors is doubled. This law has been more or less respected since the last 20 years, but the performance of memory has not known such improvements. Thus accessing the main memory is becoming more and more costly in terms of processor cycles. So each off-chip access reduces the overall efficiency of the processor because it has to wait for data to return from memory. Even if the cache hierarchy is very efficient most of the time, we still don't know how to get rid of the effect of theses misses.

One possible strategy is to use a load hit/miss predictor in order to predict early in the pipeline if a load instruction will hit or not in the cache. Based on this prediction, we can adapt the behavior of the processor to reduce the effect of potential misses. Several uses of this scheme are presented by Musoll and Lang [7], but they only talk about prediction for the L1 cache, while in our study we concentrate on the last-level cache, which is often the L2 cache in modern processors. Since L2 misses take more time than L1 misses, they are most costly for the processor and thus, it seems more beneficial to work at this level.

Several uses for the L1 cache described by Musoll and Lang can be adapted for the L2 cache but the more interesting use is for Simultaneous Multi-Threading (SMT). SMT is an architectural technique design to increase the use of the different part of the processor. Basically, it means that the processor can run two threads at the same time. But several resources of the processor are shared and thus if one thread is waiting because of an off-chip miss, these resources are reserved by the thread but are not used by it because its execution is stalled. But if we can predict these misses, we can give the entire processor to the not stalled thread and thus increase the overall efficiency of the processor.

As prediction is never 100% accurate, we need to know what are the mistakes made by the predictor. We define a hitmisprediction as a misprediction on a hit, i.e. a miss has been predicted but it is actually a hit, and a missmisprediction as a misprediction on a miss, i.e. a hit has been predicted but it is actually a miss.

Contrary to branch prediction, the fact that we have two types of misprediction induces a difficulty to exploit the results. Indeed we need to find some metrics that are able to accurately show the performance of the predictor. But as there is a prevalence of the hits, we can not just take the average of the number of hitmispredictions and missmispredictions. We choose to use four metrics also used in diagnostic testing [2].

As there is a lot of hits, it is often hard to predict misses accurately. This is why simple designs have a lack of performance in miss prediction and thus we need complex designs to achieve reasonable performance. But as designs become complex, the area cost increases and we have to make a trade-off between performance and area cost based on the potential contribution on the performance of the overall system.

This report is organized as follows. First we describe the tools we use to make this study. Then we present and analyze several predictors from the literature. After that we present the different improvements we try to make. Finally we conclude in the last section.

2 Evaluation environment

2.1 Benchmarks

To evaluate the load hit/miss predictors, we perform simulations using several benchmarks taken from the SPEC CPU2006 [1] workload. In Table 2, we present the different characteristics of the benchmarks we use. We first worked with 18 benchmarks but the floating point benchmark *povray* has not enough misses in the L2 cache to be interesting for our study. On the 17 other benchmarks, 11 are integer benchmarks and 6 are floating point benchmarks.

2.2 Simulations

In order to test the different predictors, we need a simulator to implement them. We also need to run this simulator to execute the benchmarks.

One problem of simulation is the execution time : some benchmarks can take more than one week for only one simulator run. And as we have to do hundreds of runs, we need an alternate solution. This is why we use a set of simpoints [3] to have a representation of a benchmark shorter than the complete one but that still well represents the type of work performed by the benchmark. Constructing the simpoints consists of having the profile of execution of the benchmark, i.e. see what interval of the program correspond best to the overall program behavior. Only these intervals are simulated. We set the size of one interval to 100 millions of instructions. Once we compute the simpoints, we create checkpoints at the beginning of the selected intervals to run them later.

We implement all the predictors in the SimpleScalar simulator with the Alpha instructions set. As presented in Table 1, we choose to set a configuration that can be found in modern microprocessors.

Cache	Size	Number of sets	Block size	Associativity	Replacement Policy
Data L1	32 KB	256	32	4	LRU
Data L2 (unified)	1 MB	2048	64	8	LRU

Table 1: Configuration of the caches

2.3 Exploitation of the results

One difficulty of this study is the fact that there is a strong bias towards the hits in the results, so we have to find efficient metrics to report the results. We choose four metrics also used in diagnostic testing [2]. Two of them show the results for the hits and the other two the results for the misses. They are a representation of the coverage and the accuracy of the predictors, i.e. how many hits/misses we can predict on the total hits/misses and how many errors we make on the predictions. The four metrics are:

- Sensitivity (SENS), coverage for the hits :

$$\frac{number\ of\ hits\ correctly\ predicted}{number\ of\ hits}$$

- Predictive value of a Positive Test (PVP), accuracy for the hits :

$$\frac{number\ of\ hits\ correctly\ predicted}{number\ of\ hits\ predicted}$$

- Specificity (SPEC), coverage for the misses :

$$\frac{number\ of\ misses\ correctly\ predicted}{number\ of\ misses}$$

- Predictive value of a Negative Test (PVN), accuracy for the misses :

$$\frac{number\ of\ misses\ correctly\ predicted}{number\ of\ hits\ predicted}$$

For these metrics, higher is better. But we can not have 100% in all the metrics because for that we need to have no hitmisprediction and no missmisprediction which is nearly impossible. For example, we can increase the miss coverage by predicting more misses by we lose performance in miss accuracy and hit coverage (as we predict more misses, we predict less hits). To increase all the metrics, we need to reduce at the same time the number of hitmispredictions and the number of missmispredictions.

3 Analysis of the benchmarks

Different benchmarks have different behavior in terms of cache misses, and to try to understand this miss behavior, we plot what we call the miss profile. The miss profile of a benchmarks is the accumulative fraction of misses as a function of the bias of the loads, which is the fraction of time the load misses. This kind of graph is useful to know what type of miss behavior a benchmark mainly have. A benchmark is easy to predict if its loads have a very periodic behavior. For example, if a majority of the misses are caused by always-miss loads (bias=0), such a behavior is easy to record and thus we can make accurate predictions. On the other hand, if the miss profile of some benchmarks shows that nearly all the misses are caused by loads with many different biases, then we have to identify many different behaviors and thus it is hard to predict accurately. An even more difficult situation arises if an important fraction of the misses is generated by loads that nearly always hit (bias > 0.99), as we need to predict very rare events. Moreover, a weakness of such representation is that we only know the bias of the loads and thus only the global behavior of the loads over the whole execution of the benchmarks. But there are many different behaviors that have the same bias in the end, so we can not have an idea of the precise miss behavior of the benchmarks. Figures 1, 2 and 3 show the miss profiles for the different benchmarks we use for the L1 and L2 caches.

Benchmarks	Type	Inst	Simpts	Miss rate		L2 Misses	Category
				L1	L2		
400.perlbench	INT	488	8	1.51	0.02	5428	Programming Language
401.bzip2	INT	384	17	6.13	2.06	362864	Compression
403.gcc	INT	113	19	5.99	2.19	389076	Compiler
429.mcf	INT	344	24	34.47	22.17	6559246	Combinatorial Optimization
445.gobmk	INT	287	17	0.94	0.09	21627	Artificial Intelligence
456.hmmer	INT	2498	9	0.68	0.01	2889	Search Gene Sequence
458.sjeng	INT	2941	26	0.47	0.11	28334	Artificial Intelligence
462.libquantum	INT	1925	20	48.01	24.01	2793777	Physics/ Quantum Computing
464.h264ref	INT	602	18	0.96	0.18	60093	Video Compression
471.omnetpp	INT	716	13	10.24	5.84	1311045	Discrete Event Simulation
473.astar	INT	937	18	4.15	0.62	174973	Path-finding Algorithms
410.bwaves	FP	2631	20	9.61	4.57	1361084	Fluid Dynamics
433.milc	FP	1053	19	10.24	5.87	1742180	Physics/ Quantum Chromodynamics
435.gromacs	FP	2352	19	3.88	0.18	43006	Biochemistry/ Molecular Dynamics
437.leslie3d	FP	1869	22	13.92	4.50	1325800	Fluid Dynamics
444.namd	FP	2204	20	3.87	0.03	7311	Biology/ Molecular Dynamics
453.povray	FP	1193	19	4.94	$8.94 \text{ x} 10^{-6}$	2	Image Ray-tracing
470.lbm	FP	1105	13	13.18	6.59	897780	Fluid Dynamics

Table 2: Work-set of benchmarks : name, type, number of instructions in billions (Inst), number of simpoints (Simpts), miss rates for the L1 cache (L1) and for the global cache hierarchy (L2), number of misses in the L2 cache (L2 Misses) and category of the benchmarks.

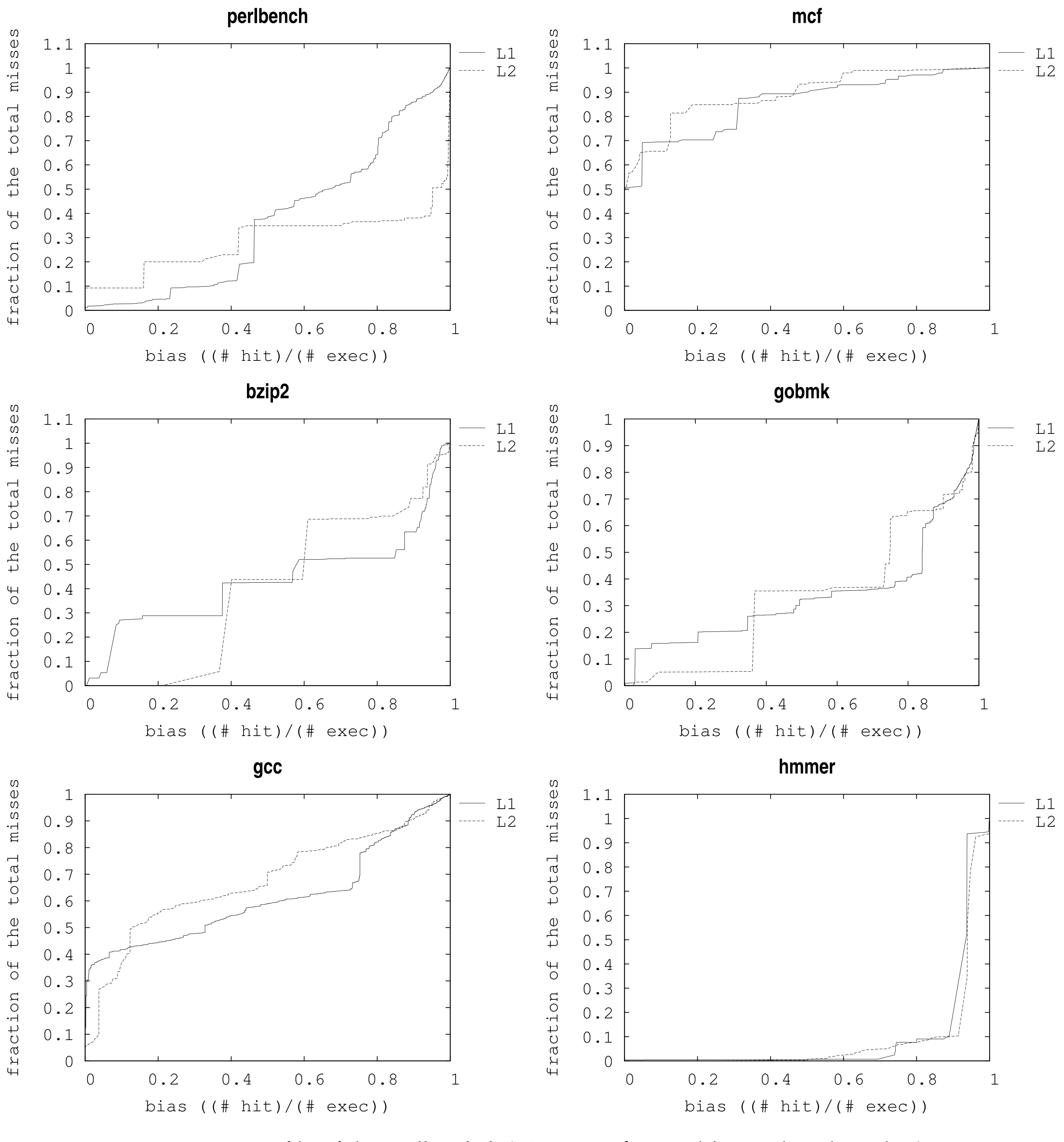


Figure 1: Miss profile of the *perlbench*, *bzip2*, *gcc*, *mcf*, *go* and *hmmer* benchmarks (integer benchmarks)

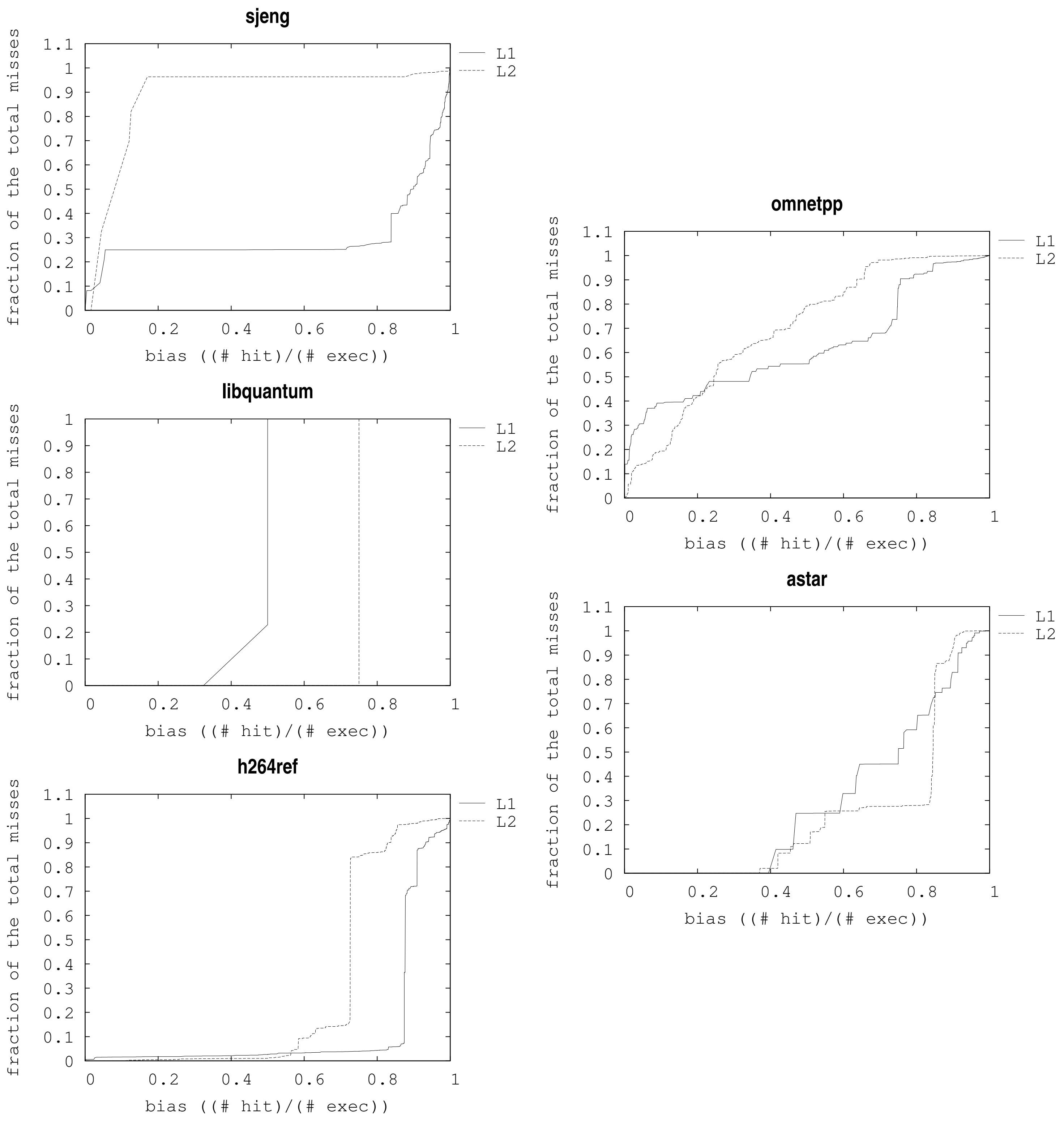


Figure 2: Miss profile of the *sjeng*, *libquantum*, *h264ref*,*omnetpp* and *astar* benchmarks (integer benchmarks)

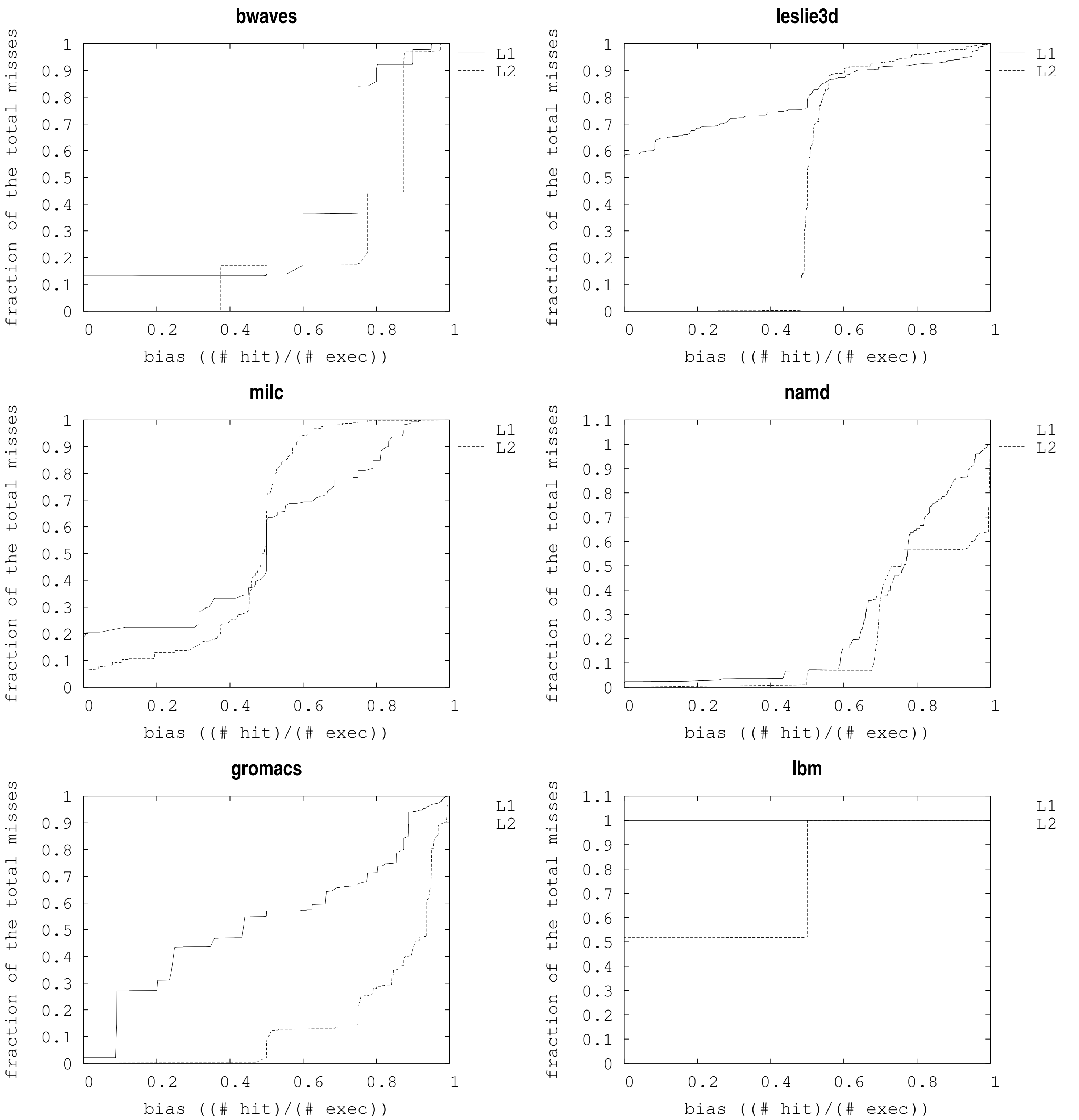


Figure 3: Miss profile of the *bwaves*, *milc*, *gromacs*, *leslie3d*, *namd* and *lbm* benchmarks (floating point benchmarks)

The integer benchmarks have a lot of loads that are difficult to predict because they don't have a full biased behavior: except for *mcf*, few loads are always-miss loads. In the case of *mcf*, we can see that 50% of the loads are always miss loads, but the other 50% are constitued of loads that are as difficult to predict as for the other benchmarks. An other exception is *libquantum*, which seems to have a very recurrent behavior as nearly all of its misses have a bias of 0.75. And thus, if this global behavior is due to some loads that periodically miss, then these misses are easy to predict.

The floating point benchmarks are on average relatively simpler to predict because they have a lot of loads which always have the same behavior. For example, *lbm* has 50% of its misses that are always-miss and 50% that have a bias of 0.5. In the case of *bwaves*, we can also see some potential regularity in the miss behavior. On the contrary, *namd* has nearly 40% of its misses that have a bias of nearly 1. It means that many loads miss only a few times, but are still responsible for 40% of the total misses.

4 Load Hit/Miss predictors

We study several predictors from the literature, from the most simple to complex history-based predictors. First, we describe the simple predictors that are not very accurate but very basic in their concept. And then, we present more complicated predictors mostly based on history of hits and misses. Most of these predictors have been studied by Musoll and Lang [7]. We also study the Limousin predictor, created by Limousin et al. [4]. The area cost for all the predictor with tables is 4 Kbit, which is a realistic number for this kind of predictor. The results of this study are presented in Figure 4. We show the results for each benchmark and an average across all benchmarks.

4.1 Simple predictors

The four simplest predictors are the Always Hit, Always Miss, Saturated counter and Bimodal predictors.

The design of the Always Hit predictor comes from a simple observation: thanks to the good properties of the cache hierarchy, the miss rate is generally low. Thus always predicting a hit is a good choice if the cost of a missmisprediction is not high. On the contrary, if the missmisprediction penalty is high, we can choose to always predict miss and use the Always Miss predictor. The results of these predictors are not presented in Figure 4 but can be derived form Table 2.

The Saturated counter predictor is derived from the implementation in the Compaq Alpha 21264 [5]. It consists of a 4-bit saturated counter that tracks the hit/miss behavior of the loads. The counter is incremented by one on a hit and decremented by two on a miss. The prediction is done looking at the high order bit of the counter: predict hit if it is 1, predict miss otherwise.

As presented in Figure 4, this predictor has a very good accuracy and coverage for the hits (except for *mcf*) but for the misses, the results are low. In fact, the counter is nearly always saturated towards hits and misses have to happen in burst if we want to be able to detect and predict them accurately. The update policy decreases the counter by two in order to reduce this effect. But still, some pattern like one miss, several hits, one miss, several hits can't be detected because the prevalence of hits drives the counter into a saturated "predict-hit" state. Moreover, only recording the global history of hits and misses is not sufficient, we need to catch this kind of information more precisely.

One way to do this is by recording hits and misses for specific situations, e.g. when executing specific load instructions. Thus, in the Bimodal predictor we try to catch the past behavior of each load instruction. This predictor is the same as the Bimodal branch predictor [8]: a table of saturated counters indexed by the PC of the load instruction. Here, saturated counters are also used but we record only local information as nearly each load has its own counter (if the size of the table is sufficient). Thus the results are much better: we have on average the same accuracy and coverage on the hits than for the Saturated counter predictor but we have a significant gain in terms of accuracy and coverage on the misses. So we see that making predictions based on the precedent behavior of a load instruction is efficient. But as we index the table with the PC of the load, aliasing can occur. Moreover, we can only predict accurately if the load has nearly always the same behavior over the execution. Thus to make better predictions, we have to record even more precise information.

4.2 History-based predictor

The Bimodal predictor recognizes that different load instructions have different hit/miss behavior. The next step is to recognize that a single load instruction is not always a hit nor always a miss. Hereto, we record the listing of hits and misses. The Per-address predictor is organized in two levels: the first level is indexed by the PC of the load instruction (like in the Bimodal predictor) and in each entry, we store H bits of the load hit/miss history (1 represents a hit, 0 represents a miss). The second level is composed of 2^H 2-bit saturated counters and we index this table using the H bits of the history. By this meaning, we can accurately predict the load if it has a periodic behavior. Indeed, each time we see the same history we map to the same entry in the second level table and use the same counter to make the prediction. If the behavior of the load is always the same then the counter is in the good saturated state.

The results for this predictor are nearly the same as for the Bimodal predictor in terms of hits. For the misses, we see that we have a little less coverage for some benchmarks but, on average, it is better, and moreover, we gain a lot in accuracy. That means that the history we record brings more precise information and the predictor is able to use that information. But here we still have the aliasing problem in the first level table and moreover we can have aliasing in the second level table. Thus, even if we catch the correct information in the tables, we can lose it before it is useful because of aliasing.

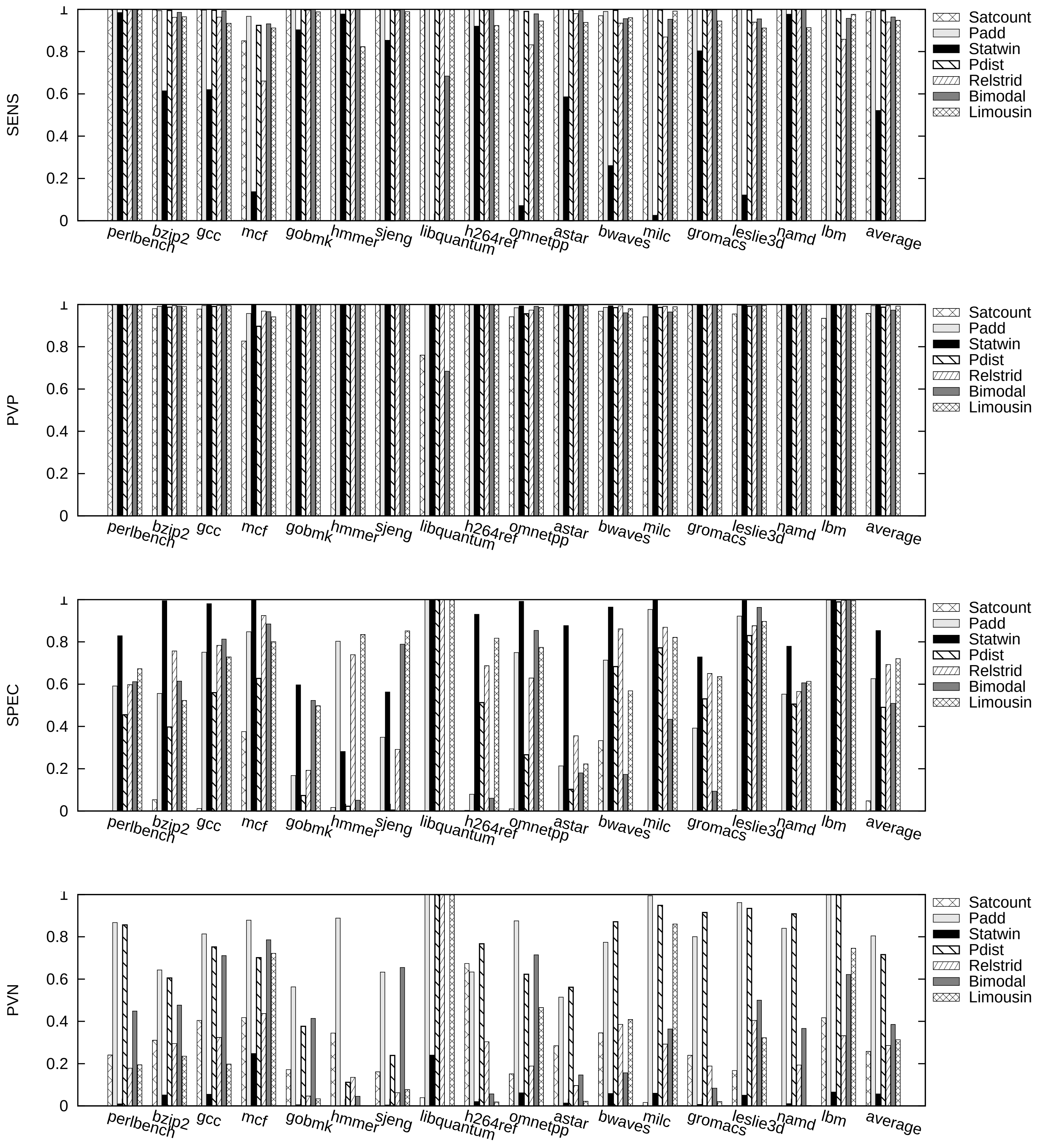


Figure 4: SENS, PVP, SPEC and PVN for the state-of-the-art predictors : a saturated 4 bits counter (*Satcount*), a per-address two level with 10 bits history and 256 entries (*Padd*), a static window with a window of 256 (*Statwin*), a per-distance two level with 10 bits history and 256 entries (*Pdist*), a per-distance relaxed strides with 256 entries and an offset of 4 (*Relstrid*), a bimodal with 2048 entries and 2 bits counters (*Bimodal*) and a Limousin predictor with 256 entries (*Limousin*).

The Limousin predictor is another predictor with a table indexed by PC. In each entry, there are two counters : one counting the number of hits between the last two misses and one counting the number of hits since the last miss. To make the prediction, we read the two counters and if they are equal, we predict a miss else we predict a hit. This predictor is designed for tracking a periodic behavior of the load with a period without too many variations. Limousin et al. [4] also put a tag for each entry, to avoid aliasing in the table, but we remove them to have nearly the same cost in area as the other predictors. We also set the number of bits of the counters, so they are considered as 8-bit saturated counters.

The results for this predictor are better than for the Bimodal predictor but are worse than the Per-address two-level. We can see it has a good miss coverage but it is at the cost of the miss accuracy and the hit coverage. This means that this predictor predicts too many misses on average.

The concept of counting the number of hits between misses is well developed by Musoll and Lang [7]. They call it "distance" and they have constructed several predictors based on this principle.

They make the observation that most of the misses occurs at short distances, i.e. the misses occur in bursts. So a simple way to use that is to predict a miss if the distance is not too high. This is how the Static-window predictor works: under a certain distance, called the window size, a miss is predicted, and over this distance a hit is predicted.

Depending on the size of window, the predictor can accurately predict all the hits. This is why we have a very good coverage. But for example, for a window size of 256, there is not always so much misses in the window, so we will do a lot of hitmispredictions just to predict all the misses. Thus the accuracy in terms of misses and the coverage in terms of hits are quite low.

After that finding, they have tried to construct a predictor on the model of the Per-address two-level predictor: the Per-distance two-level predictor. They do exactly the same thing as for the Per-address predictor but instead of indexing the first table with the PC of the load instruction, they index it by the current distance. This way they can record specific information for each distance.

The results are better than the Static window predictor so that means that this way of recording information is efficient. But if we compare with the Per-address predictor, although it has the same performance for the hits, the Per-address is nearly always better than the Per-distance for the misses. Thus it seems that using a global information (the distance) is not the best way to build a local history information.

The last predictor constructed on the distance concept is the Per-distance Relaxed Stride predictor. It is composed of a table indexed by the distance, and in each entry of this table, there are two counters, called stride. The counters record the same thing as in the Limousin predictor, so these two predictors are quite comparable in their design. But Musoll and Lang use an offset for the comparison between the two counters : they predict a miss if the value of the second counter is between the value of the first one plus the offset and the value of the first one minus the offset. By this, they have a way to predict a miss when they are at a distance where the access is likely to be a miss based on what they have previously recorded.

The results are not very good for the hit coverage compared to other predictors with tables. As they have a relaxed condition to predict miss, they actually predict more misses and increase the miss coverage, but at the cost of the miss accuracy. If we compare with the Limousin predictor, the Relaxed stride predictor has nearly the same miss coverage but the difference in accuracy is more important, so the Limousin predictor is better. This confirms our first observation on the usefulness of using the distance as a prediction parameter.

We can notice that all the results of the predictors that record history are very good on the *libquantum* benchmark. One explanation is that nearly all the misses of this benchmark have a bias of 0.75, so it is possible that these misses have a very regular behavior and thus a history-based predictor can easily catch this behavior and make accurate predictions. On *lbm*, we can observe similar results we can explain the same way.

In conclusion, for making predictions for the L2 cache, the Per-address scheme seems the more efficient. Although Musoll and Lang present the Per-distance scheme as interesting for the L1 cache, for the L2 cache the performances are not comparable with the Per-distance scheme. And within the predictors that use addresses, the Per-address two-level predictor is the best. This is why we work on the Per-address two-level predictor.

5 Improvements and results for the Per-address predictors

From the original design of the Per-address predictor we try to do several modifications to improve its performance. Some modifications are useful and bring additional performance but for some others either the analysis is incomplete at this time or the results are not significant. We present only the results for the miss accuracy and coverage because for the hits, the results are nearly always the same for a large majority of the benchmarks. Indeed, except for *mcf* which has results between 0.95 and 0.97 for the hit coverage and between 0.93 and 0.96 for the hit accuracy, the results are above 0.99 for the hit coverage and accuracy.

5.1 Improvements

5.1.1 Size of the first level table

The first level table of the two-level predictor is indexed by the PC of a load instruction. The first improvement we can make is to increase the size of this table in order to reduce as much as possible the number of load instructions mapped to one entry. Figure 5 shows the results for different size of the first level table. We see that on average, the performance increases with the size of the table, until a saturation due to the finite number of the load instructions. In some cases, we can observe some special results. For *gobmk* and *sjeng*, we see no saturation between 2048 entries and 16384 entries and so we can potentially gain more by increasing the number of entries. For *hmmer*, aliasing in the first level table seems very destructive when the number of entries is small. And finally, even if *namd* has the same results as the average for the coverage, for the accuracy, it seems that having less entries induce a constructive aliasing.

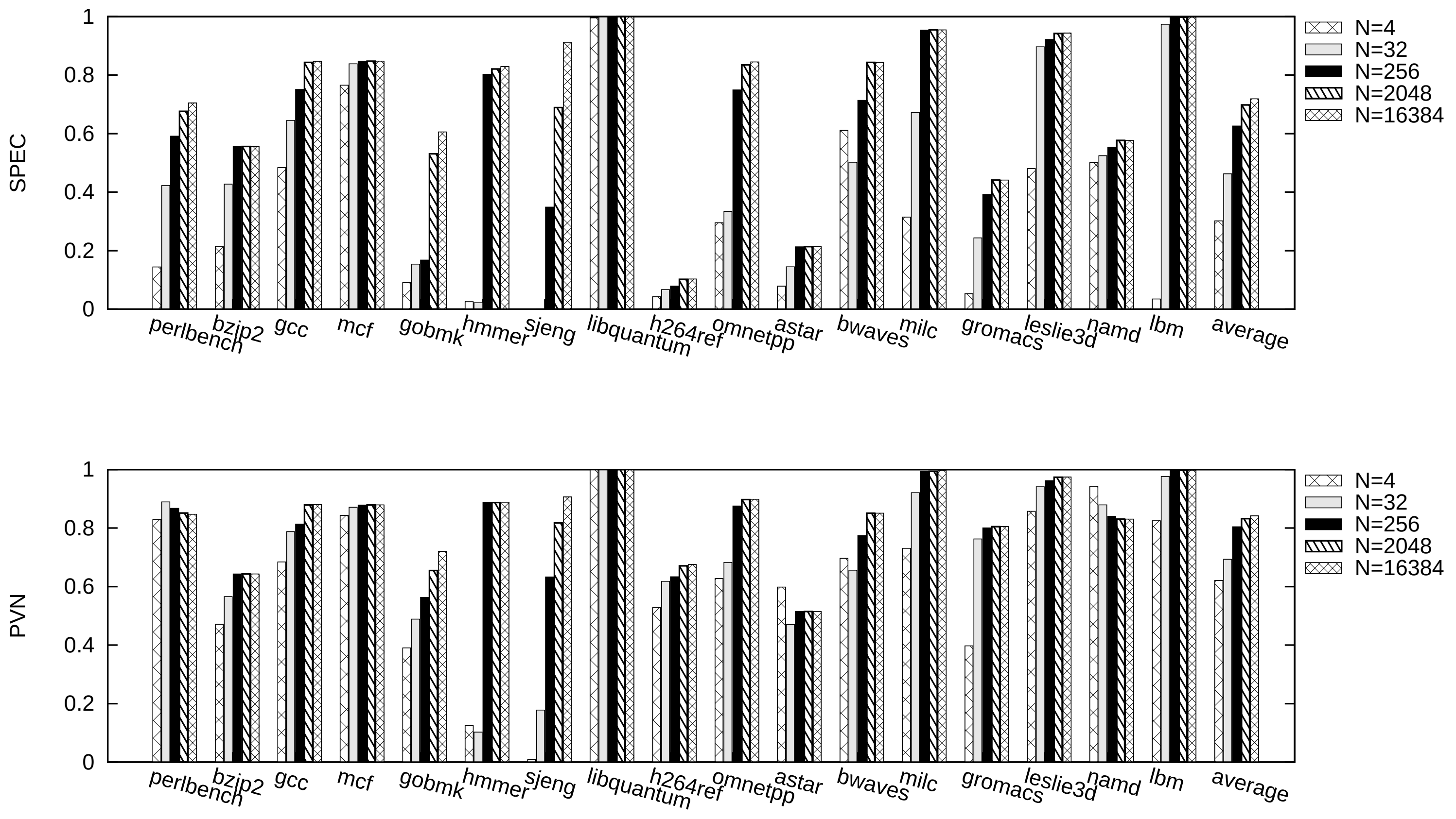


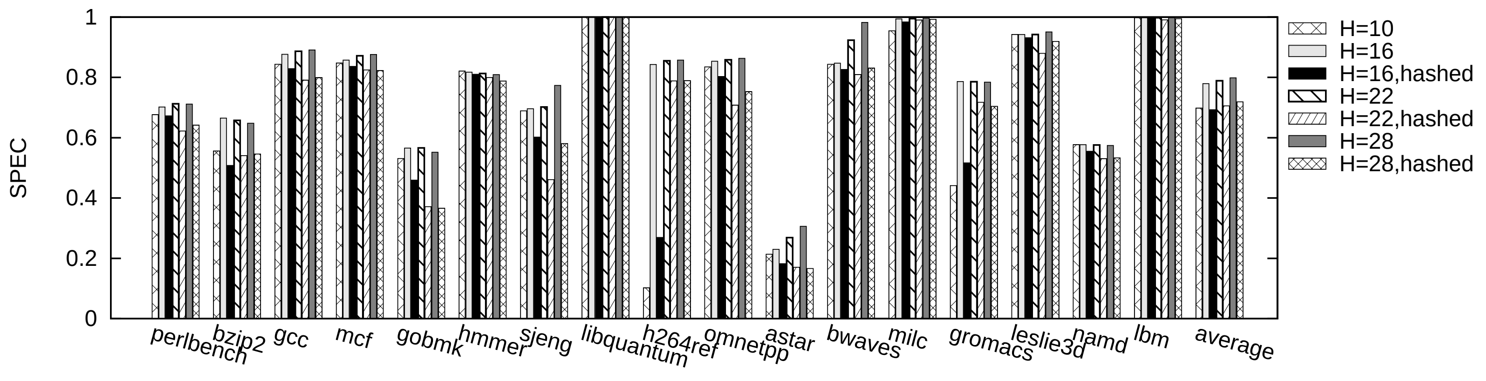
Figure 5: Evolution of the performance with the size of the first level table in the two-level predictor with an history size of H=10

In summary, increasing the number of entries in the first level table bring better performance for all benchmarks and taking a size between 256 and 2048 entries seems a good choice for the majority of the benchmarks.

5.1.2 Size of the history

In each entry of the first level table, we record the local history of the load instruction mapped to this entry. By increasing the size of this history, we can catch more precisely the behavior of the load instruction. But as we increase the size of the history, we exponentially increase the size of the second level table. To keep a realistic cost for our predictor, we can not increase too much this size. Thus we use a hash function to fold the history to 10 bits before accessing the second level table. The results are presented in Figure 6. On average, increasing the size of the history appears to be beneficial especially between 10 and 16 bits of history.

For some benchmarks, like *bzip2*, *h264ref* and *gromacs*,this effect is particularly visible but there is nearly no change in the results for the other history size. One explanation is that 16 is the number of times we can put 4 bytes in a cache line. So if a



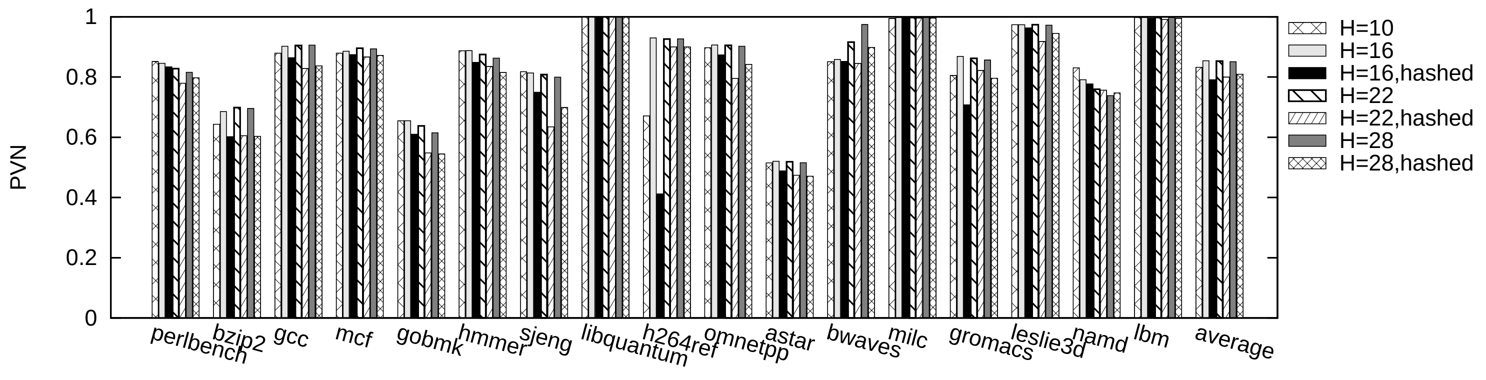


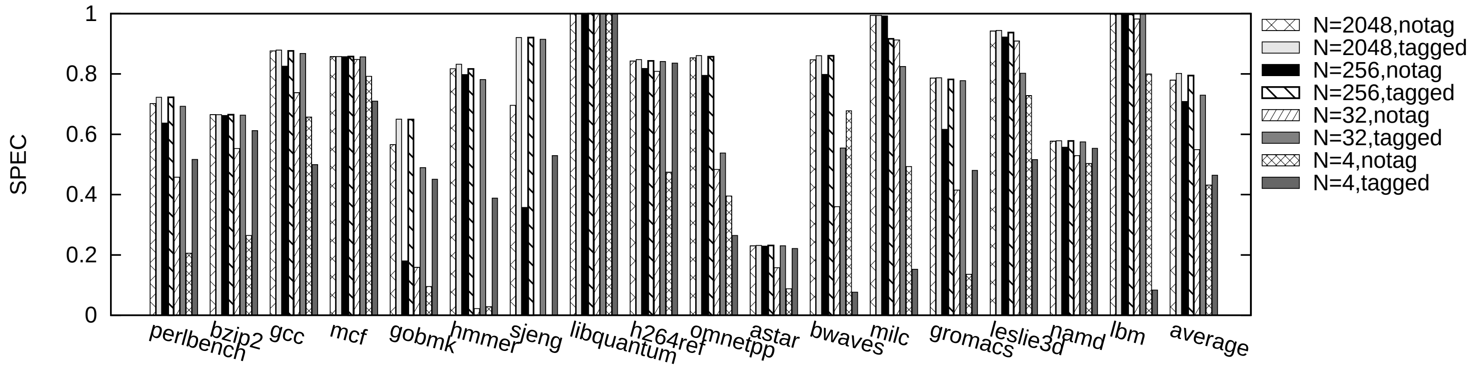
Figure 6: Evolution of the performance with the size of the first level table in the two-level predictor for a number of entries N=2048

load instruction loads a table of integers, for example, this load will make 1 miss and 15 hits after. And by having 16 bits of history, we can catch this behavior. For some other benchmarks, like *sjeng*, *astar* and *bwaves*, the performance still increases with more than 16 bits of history. A third category of benchmarks is the one where history size has nearly no influence on the results. This means that the history is not able to catch the true behavior of the load instructions of the benchmarks, or we still don't have enough bits of history to record it properly. We also see that increasing the size of the history with hashing does not bring any improvement. So hashing destroys all the new information we record with more history bits by creating more aliasing in the second level table. The benchmarks from the first two categories are the most affected by hashing because of the usefulness of the complete history. For the third category, hashing decreases the results but not dramatically. On average, it seems more useful to keep an history of 10 bits than increasing this number and hashing to have a second level table with a reasonable size.

These results show that increasing the size of the history we record seems, on average, quite efficient and, for some benchmarks like *bzip2*, *h264ref* and *gromacs*, a size of 16 bits boosts the performance. Unfortunately, We still have the problem of the exponential cost of increasing the number of entries in the second level table, as hashing is not an efficient solution.

5.1.3 Tags

The first improvements we make are only changing the parameters. With tags, we begin to change the design of the predictor. We take the idea of adding a tag in each entry of the first level table from Limousin et al.. We also upgrade the update policy in order to change the tag in an entry only if the load instruction we record is a miss. This policy tries to reduce the effect of the prevalence of hits, so we keep more information about the misses in the table. By adding a tag in the first level table, we try to reduce the aliasing in this table. The results presented in Figure 7 show several things. First, with a table as large as 2048 entries, the effect of the tags is nearly null and thus it means that for most of the benchmarks there is no aliasing with this number of entries. *gobmk* and *sjeng* are the two benchmarks where the tags are useful because they are the two benchmarks that still have a gain when we increase the number of entries from 2048 to 16384 (cf §5.1.1).



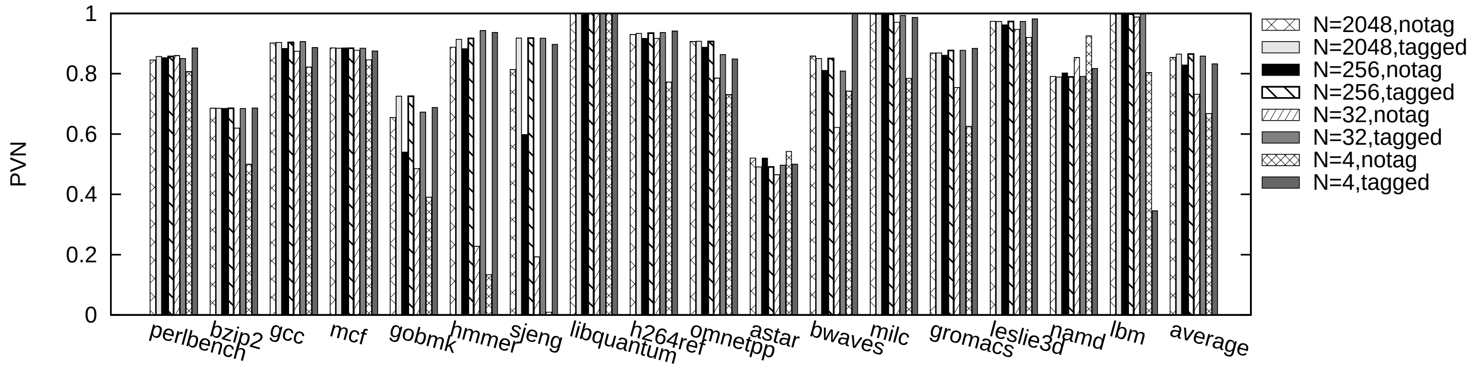


Figure 7: Evolution of the performance with or without tags for a number of entries of N=2048, N=256, N=32 and N=4 and for an history size of H=16

We also see that we can reduce the size of the first level table to 256 entries without losing performance. It means that having a tag effectively reduce aliasing in the first level table. The only exception is for *milc* where the miss coverage decrease with the tags. It can be due to some ping-pong phenomena between two load instructions which miss often and that are mapped to the same entry. Thus we make more miss-misprediction and have less miss coverage.

We can even reduce the size of the table to 32 entries: we lose performance but we are still better on average than with 256 entries and no tag. For *omnetpp*, we begin to see the destructive effect of the ping-pong phenomena. These phenomena are even stronger when we decrease the number of entries to 4. At this point, even if we keep good performance for the miss accuracy on average, we lose a lot in coverage. Moreover, for some benchmarks like *gcc*, *omnetpp*, *bwaves*, and *lbm* the coverage with the tags is really low compared to without tag. As accuracy increases during the same time, that means we predict less misses so the coverage decreases, but we predict them well.

Finally, having a tag in the first level table nearly avoids aliasing and thus we can reduce the size of the table from 2048 to 256 entries without losing performance. Moreover it is more cost effective to have less entries with a tag than more entries without tag. But we still have to keep a certain number of entries and maybe use associative tables to avoid ping-pong phenomena.

5.2 Other experiments

We try to implement some other ideas but these modifications do not bring any improvements and so we do not show results, we only discuss possible explanations about why such modifications are not useful and what we can infer on the benchmarks behavior.

5.2.1 Number of bits of the counters

As the Per-address two-level predictor uses saturated counter to record some behavior, we try to see what is the effect of changing the number of bits of these counters. As we increase this number, aliasing has less impact on performance. A side effect is that it takes more time to record a new behavior, but we can reduce this effect by adapting the update policy of the counters, like in the Saturated counter predictor. There is nearly no improvement in the results when we replace the 2-bit saturated counters with 3 or 4-bit saturated counters. Even for some benchmarks, there is some loss in performance. This means that either there is no significant aliasing in the second level table or increasing the number of bits of the counters is not sufficient to reduce it. We still have to work on it to know the impact of aliasing in the second level table.

5.2.2 Filtering of the entries

A difference between load prediction and branch prediction is the bias of the outcome. To reduce the effect of the prevalence of hits, we put a simple Bloom filter before the access to the predictor. This Bloom filter is a simple table indexed by the PC of the load instructions and where we keep a counter for each entry that counts the number of time the load misses. If this number is above a threshold, then we use the predictor to make prediction. If not, we simply predict a hit. By this construction, we hope to effectively reduce the number of hits we deal with in the predictor and thus having less biased information. But the results are not as we expect, and having a Bloom filter change nothing to the behavior of the predictor. Indeed, the Bloom filter is efficient to filter the nearly always-hit loads, but Figures 1, 2 and 3 show that such loads are not predominant. Moreover, in the majority of the benchmarks, most loads that miss have a bias between 0.4 and 0.8. Thus, if they miss often in the beginning and after are always-hit, the Bloom filter can not filter this type of load, so they still pollute the predictor.

To filter the load instructions, this solution, as we have implemented it, is not efficient. As we see, adding a tag in the first level table is a more appropriate solution to filter loads. When using tags, loads can be filtered from the first level table during periods of always-hit.

5.2.3 Aliasing in the second level table

We see that recording more bits of history for each entry is quite efficient, but the size of the second level table increases with each more bit. Thus we try to reduce this size by hashing the history before accessing the table. As the results with the folding hash function are not good enough, we try to use another hash function taken from branch prediction [6]. But the results for this hash function are even worse than the results with the first hash function.

This means that even with a different hash function we still have destructive aliasing. To try to understand why aliasing in the second level table destroys so much information, we analyze how often each entry of this table is used. With this kind of information, we can also try to implement a specific hash function that will not induce aliasing. But we find no apparent pattern in the entries used that explains why aliasing is so destructive.

As we can not use hashing without destroying a part of the information we record in the history, we try to change how the history is recorded to reduce the effect of aliasing. We use saturated counters to record hits and each time there is a miss we change of counter. Thus we have just one counter that is saturated when we have many hits for a long time. We try different configurations with different number of counters and different number of bits.

The preliminary results we have show that no single configuration is efficient for all benchmarks. Moreover, on average, the results are worse than with the normal history. Thus, this way of recording history seems not precise enough to make good predictions.

6 Conclusion

In this study, we have worked on load hit/miss predictors to have an idea of the performance of several predictors proposed in the literature. For that we have used four metrics to represent the hit/miss accuracy, i.e. how accurate the predictor is on predicting hits/misses, and the hit/miss coverage, i.e. how many hits/misses the predictor can predict over the total number of hits/misses.

We have shown that with these metrics, the Per-address two-level predictor seems to be the more powerful design. Then we have successfully improved its performance by increasing the size of the different levels and by adding tags in the first level table. As we have seen, tags are a good trade-off to reduce aliasing in the first level table and to filter the entries that are not useful. Increasing the size of the history is good on average and extremely important for some benchmarks. We have also seen that keeping a reasonable size for the second level table by hashing the history decreases performance.

The results show that it is possible to have good performance on load hit/miss prediction on average but for several benchmarks, the results are still low even with our improvements. Future studies can be made to better understand the importance of aliasing in the second level table and how to reduce it.

Finally, we see that the Per-address two-level predictor is a quite efficient predictor but several other designs can be constructed and studied, to maybe have load hit/miss predictors as efficient as branch predictors.

Acknowledgments

I want to thank all the PARIS team and Koen de Bochere for accepting me in his team for this internship. I also want to thank my adviser Hans Vandierendonck for this great internship and for his daily presence to answer my numerous questions and for his help to find right directions on this study. I also thank Sean Rul and Veerle Desmet for their advices and for having shared their office with me.

References

[1] SPEC CPU2006. *http://www.spec.org/cpu2006/*.

[2] D. Grunwald, A. Klauser, S. Manne, and A. R. Pleszkun. Confidence estimation for speculation control. In *ISCA*, pages 122–131, 1998.

[3] G. Hamerly, E. Perelman, J. Lau, and B. Calder. Simpoint 3.0 : Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism*, vol. 7, September 2005.

[4] C. Limousin, J. Sebot, A. Vartanian, and N. Drach-Temam. Improving 3D geometry transformations on a simultaneous multithreaded SIMD processor. In *ICS '01: Proceedings of the 15th International Conference on Supercomputing*, pages 236–245, New York, NY, USA, 2001. ACM.

[5] E. J. McLellan and D. A. Webb. The Alpha 21264 microprocessor architecture. In *ICCD '98: Proceedings of the International Conference on Computer Design*, page 90, Washington, DC, USA, 1998. IEEE Computer Society.

[6] P. Michaud and A. Seznec. A comprehensive study of dynamic global history branch prediction. *IRISA, Report No 1406*, page 20, June 2001.

[7] E. Musoll and T. Lang. Distance-based prediction of hit/miss of L1 caches. *Tech. Rep., Department of Electrical and Computing Engineering, UC Irvine*, 2002.

[8] J. E. Smith. A study of branch prediction strategies. In *ISCA*, pages 135–148, 1981.