Brief Contributions

Using Indexing Functions to Reduce Conflict Aliasing in Branch Prediction Tables

Yi Ma, Hongliang Gao, and
Huiyang Zhou, *Member*, *IEEE*

Abstract—High-accuracy branch prediction is crucial for high-performance processors. Inspired by the work on indexing functions to eliminate conflict-misses in memory hierarchy, this paper explores different indexing approaches to reduce conflict aliasing in branch-prediction tables. Our results show that indexing functions provide a highly complexity-effective way to enhance prediction accuracy.

Index Terms—Processor architectures.

1 Introduction

CURRENT microprocessor design trends expose branch prediction as a primary performance bottleneck. Higher branch prediction accuracy not only exploits higher degrees of instruction-level parallelism (ILP), but also helps microprocessors to fit in the shrinking power/energy consumption budget by wasting less execution time on wrong paths.

With its critical role in processor design, there has been extensive research on branch prediction to improve prediction accuracy. Many existing branch predictors can be grouped into two major categories: context-based and computational branch predictors. Context-based branch predictors, including two-level [17] and g-share predictors [8], employ various Markov Finite State Machine (FSM) models to capture branch behavior [2]. Computational branch predictors, such as perceptron predictors [5], [6], [16], model the branch correlation relationship as a linear combination of branch history. For either class of predictors, the prediction tables are usually indexed with a hash function of branch address (*pc*) and some local/global history (LHR/GHR) bits. With limited prediction table sizes, different branches interfere with each other through shared table entries, which are known as conflict aliasing [9]. Previous studies [9], [11], [14] showed that conflict aliases undermine prediction accuracy severely for context-based branch predictors, especially for simple hash functions such as the modulo mapping (i.e., *index = pc mod number of table entries*), where the number of entries in a table usually is a 2's power.

For computational branch predictors such as perceptron predictors, each entry in the prediction table contains multiple perceptron weights, which capture the correlation between the current branch outcome and relatively long branch history. These large entries result in prediction tables with much fewer entries than context-based predictors for the same hardware budget. Therefore, the adverse impacts of conflict aliases are more evident in perceptron predictors (see Section 5).

Conflict misses are also recognized as a performance bottleneck in memory hierarchy. Besides employing set-associative structures, indexing functions have been shown to be effective in eliminating conflict misses [4], [7], [15]. In this paper, we explore indexing functions to reduce conflict aliases in branch prediction tables. For complex indexing functions such as prime-modulo mapping, we propose expanding the branch target buffer (BTB) to amortize the computation cost. Simple indexing functions such as bitwise-XOR schemes, on the other hand, only need very limited hardware (e.g., a few XOR gates). Our experiments show that the indexing functions are effective in improving branch prediction accuracy for perceptron branch predictors (e.g., 4.5 percent on average for a 128-entry perceptron predictor) at negligible hardware costs. For context-based branch predictors with large table sizes, slight improvements (e.g., 1.6 percent on average for a 2^{15}-entry g-share predictor) are observed from the indexing functions such as the bitwise XOR mapping. Multiple indexing functions, in addition, can also improve the prediction accuracy of optimized 2bcgskew predictors (e.g., 3.8 percent on average for a 32K-bit predictor). Our results also show that the complex indexing functions, although more effective in reducing conflict misses in caches, do *not* consistently outperform the simple indexing functions on branch prediction tables and all the performance improvements can be achieved with highly cost-effective indexing schemes that require only a few XOR gates.

The remainder of the paper is organized as follows: Section 2 presents the related work on indexing functions to eliminate cache conflict misses and introduces the notations that are used throughout the paper. Latency concerns of complex indexing functions are addressed in Section 3. Section 4 explores various indexing functions in detail and Section 5 evaluates their performance impact on different branch predictors. Section 6 concludes the paper.

2 Related Work

Indexing functions were studied initially to achieve uniform access distribution across memory banks [3], [10]. The research was then extended to cache indexing in order to eliminate conflict cache misses [4], [7], [15].

In [4], various XOR-based indexing functions are evaluated on different cache organizations. For an address *addr*, which is decoded into several fields, as shown in Fig. 1, the first b bits are the block offset (*blk_off*) where b is determined as $log_2(block\ size)$, whereas for branch prediction tables, b is $log_2(instruction\ size)$. The next two k-bit fields, where

$$k = log_2(number\ of\ sets\ in\ the\ cache\ or\ branch\ prediction\ table),$$

are denoted as x and y. The traditional modulo-mapping is simply $index = x$. The bitwise XOR indexing function defines the index as the bitwise XOR of x and y (i.e., $index = x \hat{} y$).

Another XOR-based indexing function that has been shown effective in eliminating conflict cache misses is the irreducible polynomial mapping [4], [10], [15], which considers the line address A as a polynomial function $A(x) = a_{n-1}x^{n-1} + \ldots + a_1x^1 + a_0$. Then, based on modulo-2 polynomial arithmetic and an irreducible polynomial $P(x)$, the polynomial $R(x) = A(x)\ mod\ P(x)$ is the resulting index. The degree of the irreducible polynomial $P(x)$ (over GF(2)) is k (i.e., the number of index bits). The implementation of the irreducible polynomial mapping requires parallel XOR operations and does not incur overly complex circuits, as addressed in [10].

The prime-modulo hashing, defined as $index = A\ mod\ p$, where p is a prime number closest to but smaller than the number of cache sets, can achieve a more uniform cache access distribution and has been shown to be less susceptible to pathological behavior [7]. However, the computation of prime modulo involves high

• *The authors are with the School of Computer Science, University of Central Florida, Orlando, FL 32816-2362. E-mail: {yma, hgao, zhou}@cs.ucf.edu.*

Manuscript received 26 Sept. 2005; revised 22 Feb. 2006; accepted 2 Mar. 2006; published online 21 June 2006.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0331-0905.

0018-9340/06/$20.00 © 2006 IEEE Published by the IEEE Computer Society

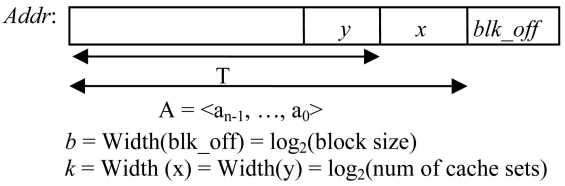

Fig. 1. Decoding an address *addr*.

hardware cost and additional latency. In addition, the fragmentation problem leads to wasted cache sets when the number of cache sets is not a prime number.

An alternative to the prime-modulo hashing, as proposed in [7], is the prime-displacement hashing and defined as $index = (p * T + x)\ mod\ (num\ of\ cache\ sets)$, where p is a prime number and T and x are address fields defined in Fig. 1. If the binary representation of p takes few 1s, the implementation only requires few shifting and truncated add operations.

Skewing [1] is a technique that uses different hashing functions to index multiple cache ways to avoid conflict misses. The irreducible polynomial mapping and prime-displacement hashing can be used to improve upon skewed-associative caches by using different irreducible polynomials or prime numbers to compute the indexes for different cache ways [4], [7], [15]. In context-based branch predictors, skewing is used to trade capacity aliases for fewer conflict aliases [9].

3 Eliminating Latency Overheads of Complex Indexing Functions

In high-performance processors, branches are predicted in a pipelined fashion. In a generic conditional branch prediction model, shown in Fig. 2, in order to fetch up to one basic-block instruction each cycle, the current branch *pc* is used to index both the branch prediction table for prediction and the branch target buffer (BTB) for path information (shown as (1) in Fig. 2). Based on the predicted direction, the information of the predicted path, including the starting fetch address, the length of the basic block, and the next branch address along the predicted path, is selected (shown as (2) in Fig. 2). In the next cycle, the next branch address will then be used as the current branch *pc* (shown as (3) in Fig. 2).

For complex indexing functions, the computation latency may exceed one cycle, which in turn undermines the throughput of the instruction fetch unit. The solution that we propose is to expand the path information field in each BTB entry to include the prediction table index of the *next* branch, as shown in Fig. 3. The prediction table index is only computed at the BTB replacement when the next branch address information is collected. Then, when this BTB entry is reaccessed, the next branch index field will be

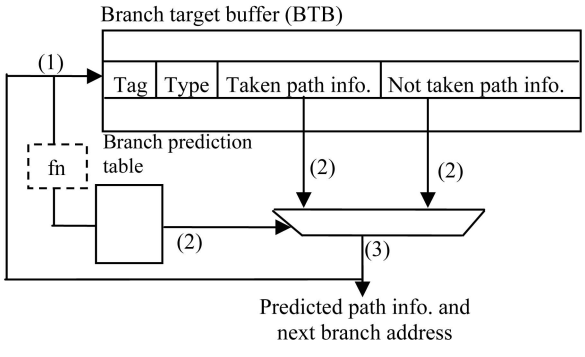

Fig. 2. A generic branch predictor model with an indexing function *fn*.

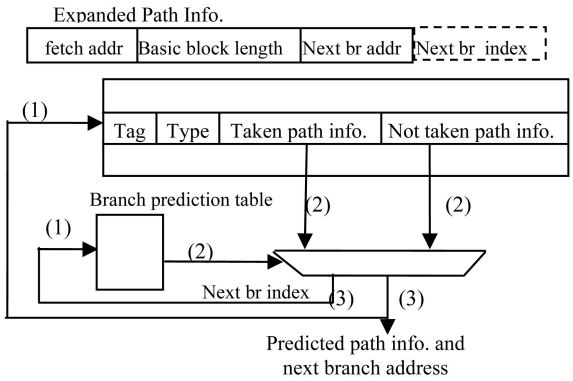

Fig. 3. Expanding the path information in BTB to store complex index computation.

used directly to index the branch prediction table for next branch prediction (shown as (3) in Fig. 3). In this way, the computation latency of the indexing functions is effectively shifted off the branch prediction critical path. In addition, such a prestored index also enables the use of a prediction table with any number of entries (i.e., not necessary a 2's power). Note that the next branch index field is accessed at the same time as the next branch address field (shown as (2) in Fig. 3). Therefore, there is no associated latency penalty. Although appending the prediction table index into the BTB incurs additional storage cost, it does not significantly affect the critical access latency of prediction tables. Our experiments using CACTI 3.2 [13] show that adding 24 extra bits into an 8-byte data block incurs only 3.5 percent access time penalty for a 2k-entry 4-way BTB using 90nm technology.

For simple indexing functions such as the bitwise XOR mapping, there is *no* need to expand BTB entries. As the XOR operations on each bit can be performed in parallel, the extra latency is only one XOR gate, which is unlikely to affect the prediction critical path.

4 Indexing Branch Prediction Tables

Although using indexing functions to eliminate conflict aliasing in branch prediction tables is inspired by the indexing functions for caches, there exist major differences between cache indexing and branch prediction table indexing. First, adding set associativity incurs the cost of tags, especially for context-based predictors, which usually feature a large number of table entries to explore branch correlation. Second, in branch prediction tables, each entry captures certain branch history behavior. Therefore, it is important that, once the table entries are allocated for a particular branch, they will be accessed by the same branch even with the set-associative organizations to get previously learned history knowledge. This incurs additional constraints compared to set-associative caches since the common replacement policy, such as LRU, cannot maintain such placement consistency. Third, while an aliasing in caches definitely results in a cache miss, an aliasing in branch predictor tables does not necessarily incur a misprediction. Therefore, indexing functions such as prime-modulo mapping that are effective in reducing conflict cache misses may not work well with branch prediction tables (see Section 5).

Next, we explore various indexing functions for different branch predictors. We first focus on branch predictors using single hashing functions, including g-share and perceptron predictors. Then, we address multiple indexing functions using 2bcgskew branch predictors.

4.1 Single Indexing Functions

For conciseness, we assume a branch prediction table with 2^k entries. For a table with an arbitrary number of entries, the index can be computed at BTB replacement and stored in the BTB as discussed in Section 3.

4.1.1 Modulo Mapping

This traditional indexing function defines index as: $Index = x = A \bmod 2^k$, where x and A use the same notations as in Fig. 1. In a g-share predictor, A is $pc[31:2]^\wedge GHR[k-1:0]$ (assuming the instruction size as 4 bytes and a 32-bit pc) and, for perceptron predictors, A is simply $pc[31:2]$.

4.1.2 Bitwise-XOR Mapping

The bitwise-XOR indexing function defines the index as $Index = y^\wedge x$, where y and x are defined as in Fig. 1. For g-share predictors, the bitwise XOR index is $pc[2k+1:k+2]^\wedge pc[k+1:2]^\wedge GHR[k-1:0]$ and such a computation can be carried out using the same number of XOR gates as the modulo mapping, but with fan-ins increased from 2 to 3. For perceptron predictors, the bitwise XOR mapping requires k XOR gates and the index becomes $pc[2k+1:k+2]^\wedge pc[k+1:2]$.

4.1.3 Irreducible-Polynomial (I-Poly) Mapping

The polynomial $R(x)$, which is determined by $R(x) = A(x) \bmod P(x)$, is the binary representation of the index and it can be computed in the following manner [10]: $R(x) = a_{n-1}R_{n-1}(x) + \ldots + a_1R_1(x) + a_0R_0(x)$, where $R_i(x) = x^i \bmod P(x)$ can be precomputed once the I-poly $P(x)$ is selected. The implementation of the I-poly mapping requires XOR gates with multiple fan-ins. In our experiments, among all the irreducible polynomials with degree k, we select one that does not incur too many additional fan-ins to the XOR gates. For g-share branch predictors, $A = pc[31:2]^\wedge GHR[k-1:0]$, whereas, for perceptron predictors, $A = pc[31:2]$.

4.1.4 Prime-Modulo Mapping

Based on a prime number p, which is closest to but smaller than 2^k, the index is computed as $Index = A \bmod p$. Since the prime-modulo mapping only uses a prime number of table entries, it results in wasted table entries or the fragmentation problem [7], which may lead to inferior performance compared to other mapping schemes. To eliminate such inefficiency, we propose another prime-modulo scheme (referred to as ***a-prime-modulo mapping***), defined as follows: $Index = A \bmod p \bmod 2^k$, where p is a prime number that is closest to but *larger* than the number of table entries. In this way, all table entries will be utilized at a cost of slightly nonuniform access distribution since the first few entries are used more often than others.

The computation latency of the prime-modulo indexing function can be hidden by expanding BTB entries to store the prime-modulo results, as discussed in Section 3. Since such prime-modulo computation is only performed at BTB replacement, it presents additional constraints for branch predictors requiring history information such as g-share predictors. The reason is that the history information may vary after the BTB replacement. As a result, for g-share predictors, the prime-modulo indexing function is defined as $(pc[31:2] \bmod p \bmod 2^k)^\wedge GHR$ instead of

$$(pc[31:2]^\wedge GHR) \bmod p \bmod 2^k.$$

For perceptron predictors, the prime-modulo indexing function is $(pc[31:2] \bmod p) \bmod 2^k$.

4.1.5 Prime-Displacement Mapping

In the prime-displacement mapping, the index is computed as $Index = (T * p + x) \bmod 2^k$, where T and x use the same notations as in Fig. 1. The performance of the prime-displacement mapping is dependent on the selection of the prime number p. In this paper, we select p as 17 to minimize the hardware cost to implement such computation (i.e., a 4-bit shift and one truncated addition). For g-share branch predictors, $A = pc[31:2]^\wedge GHR[k-1:0]$ and, for perceptron predictors, $A = pc[31:2]$.

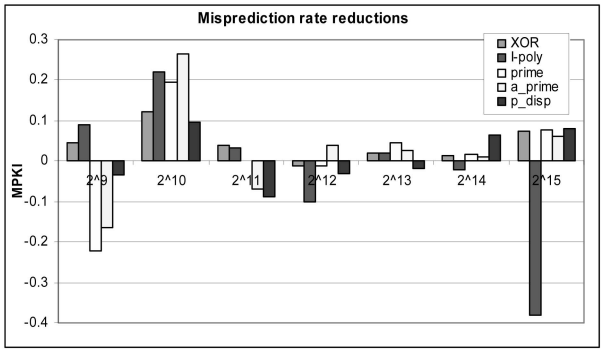

Fig. 4. Average misprediction rate reductions (the higher the better) achieved by indexing functions on g-share branch predictors with different table sizes ranging from 2^9 to 2^{15} entries.

4.2 Multiple Indexing Functions

The skewing technique, exemplified with the optimized 2bcgskew branch predictor [12], is shown to be highly effective to eliminate conflict misses by trading capacity misses. As the optimized 2bcgskew predictor has multiple branch prediction tables (one bimodal table, two g-share tables, and one meta table), we can use different indexing functions for each of them to further reduce conflict aliases.

5 Experimental Methodology and Results

To evaluate the performance impact of the indexing functions upon different branch predictors, the Championship Branch Prediction (CBP) [18] branch traces are used as they encompass representative workloads from a wide range applications, including multimedia (MM), server (SERV), integer (INT), and floating-point (FP) applications. Among them, the INT and FP benchmarks are from the SPEC CPU 2000 benchmark suite. The branch misprediction rate is measured as the number of mispredictions per 1,000 instructions (MPKI). The timing simulator with the CBP traces, unfortunately, is currently unavailable. In our experiments, we also use an ideal BTB for the complex indexing functions, including prime-modulo and irreducible polynomial functions, in order to model the best performance that can be achieved.

In our first experiment, we evaluate the single indexing functions on g-share branch predictors with different sizes. The baseline (i.e., the modulo mapping) misprediction rates on average are 13.58, 11.82, 9.89, 8.35, 7.08, 5.97, and 5.18 MPKI when prediction table sizes (not including BTB) ranging from 2^9 to 2^{15} entries (i.e., from 1K bits to 64K bits). The reductions in misprediction rates (measured by MPKI reduction, the higher the better) achieved by various indexing functions are shown in Fig. 4.

From Fig. 4, it can be seen that the performance impacts vary for different indexing functions. The simple bitwise XOR indexing function (labeled "XOR") improves prediction accuracy for all g-share configurations except for a prediction table with 2^{12} entries. For a large table of 2^{15} entries (i.e., 64K bits), it reduces the misprediction rate from 5.18 to 5.10 MPKI (i.e., a 1.6 percent improvement). Complex indexing functions, including the irreducible polynomial (labeled "i-poly"), prime modulo (labeled "prime"), and a-prime modulo (labeled "a-prime") mapping, show no consistent performance improvement over simpler ones

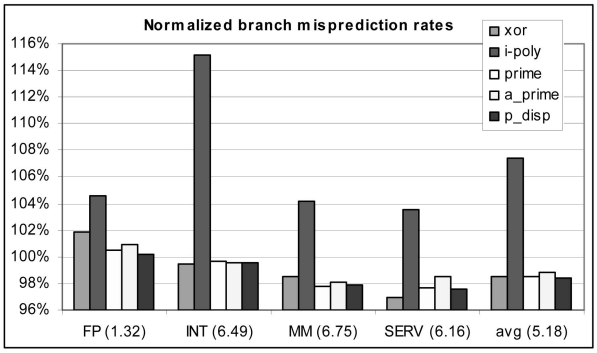

Fig. 5. Normalized misprediction rates for a g-share branch predictor with 215 entries (the lower the better). The baseline misprediction rates (in MPKI) are included for each category of benchmarks.

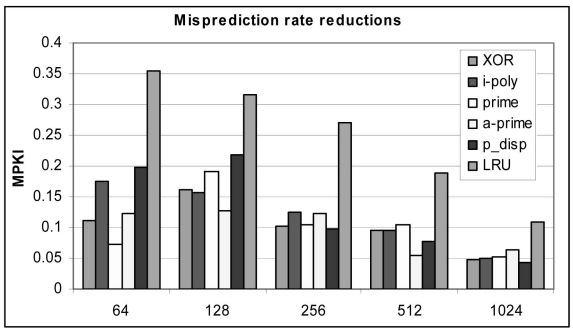

Fig. 6. Misprediction rate reductions (the higher the better) achieved by indexing functions on perceptron branch predictors with different table sizes.

and are susceptible to pathological behavior. The prime-displacement (labeled "p_disp") mapping performs better than the bitwise XOR mapping for large prediction tables while it is less effective for small ones.

A closer look at each individual category of benchmarks, which is reported as misprediction rates normalized to the baseline misprediction rates for a g-share predictor with 2^{15} entries shown in Fig. 5, reveals that the SERV benchmarks benefit most from the indexing functions as they have the largest branch working set (a misprediction rate reduction of 3 percent with the bitwise XOR mapping). For the FP benchmarks, the indexing functions increase the misprediction rates. However, since the baseline misprediction rates for the FP benchmarks are quite limited (1.32 MPKI), the impacts of indexing functions are actually very small for the FP workloads. The irreducible polynomial mapping suffers from pathological behavior and increases the misprediction rates for all types of workloads.

Overall, we can see that carefully selected indexing functions (e.g., the bitwise XOR mapping for small prediction tables and prime-displacement mapping for larger tables) can improve the g-share branch predictors slightly with negligible hardware cost (i.e., a few XOR gates for the bitwise XOR mapping or shifting and addition for the prime displacement mapping).

Next, we examine the impact of the indexing functions upon computational branch predictors, or perceptron predictors. In this experiment, we choose the perceptron predictors with 10-bit LHR and 34-bit GHR. The average misprediction rates using the baseline modulo indexing function are 6.04, 4.88, 3.96, 3.47, and 3.18 MPKI, respectively, for a prediction table with 64 (23.7K bits), 128 (47.3K bits), 256 (94.7K bits), 512(189.4K bits), and 1,024 (378.9 K bits) perceptrons.

In addition to the indexing functions discussed in Section 4, we also model an LRU placement method to highlight the performance potential of conflict reduction for perceptron predictors. In this LRU approach, each perceptron maintains a tag field and an LRU field. When a branch is replaced from the prediction table, its index is saved in a secondary mapping table, which is used to make sure that same branch will access the same perceptron when it reaccesses the prediction table. Note that this LRU method is only used to show the performance potential of eliminating conflict aliases, rather than as a practical design. The reductions in the misprediction rate resulting from the indexing functions and the LRU approach are reported in Fig. 6.

From Fig. 6, it can be seen that the indexing functions can reduce misprediction rates significantly for a wide range of perceptron predictors. Smaller size prediction tables, e.g., a table with 128 perceptrons, generally benefit more from the indexing functions than large size prediction tables, e.g., a 1,024-entry table. The reason is that the smaller tables result in a higher number of conflict aliases. All the indexing functions exhibit significant improvement on prediction accuracy. For small table perceptron predictors (64 and 128 entries), the prime-displacement mapping achieves the best performance improvement and reduces the misprediction rate by 0.20 MPKI (a 3.3 percent improvement) and 0.22 MPKI (a 4.5 percent improvement), respectively. Irreducible polynomial mapping performs well for a 256-entry prediction table, while the prime modulo and a-prime modulo mapping achieve the highest improvement for a 512 and 1,024-entry prediction table, respectively. The simple bitwise XOR scheme, on the other hand, also achieves good performance improvement for large prediction tables. Given its high cost-effectiveness, the bitwise XOR mapping would be a more appropriate design option. The LRU results shows that, even with carefully selected indexing functions, there is still a significant performance potential by further reducing conflict aliases.

The normalized branch misprediction rates for each category of workloads, as shown in Fig. 7 for a perceptron predictor with 256 perceptrons, show that the INT and SERV benchmarks benefit the most by reducing conflict aliases. Floating-point benchmarks have low misprediction rate (0.73 MPKI) and the prime-displacement mapping slightly increases the misprediction rate (0.04 MPKI) due to some pathological behavior.

Overall, the experiments show that the indexing functions can improve the prediction accuracy of perceptron predictors significantly. One complexity-effective design choice would be using the prime-displacement for small prediction tables and the bitwise XOR for large prediction tables. The cost of hardware implementation is negligible as neither bitwise XOR mapping nor prime displacement mapping requires complex implementation.

As discussed in Section 4, we use optimized 2bcskew branch predictors to study the impact of multiple indexing functions. The 2bcgskew predictor contains two g-share tables (2^{N-1} bits in total), a bimodal table, one meta table (2^{N-2} bits in total), and one shared

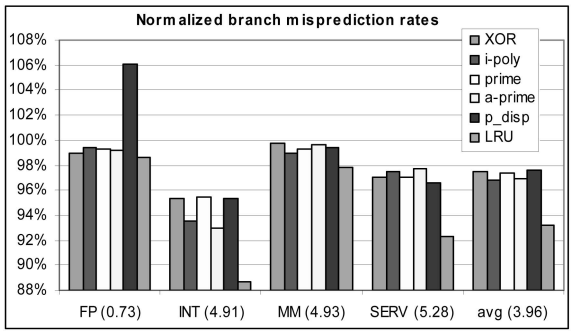

Fig. 7. Normalized misprediction rates for a perceptron branch predictor with 256 entries (the lower the better). The baseline misprediction rates (in MPKI) are included for each category of benchmarks.

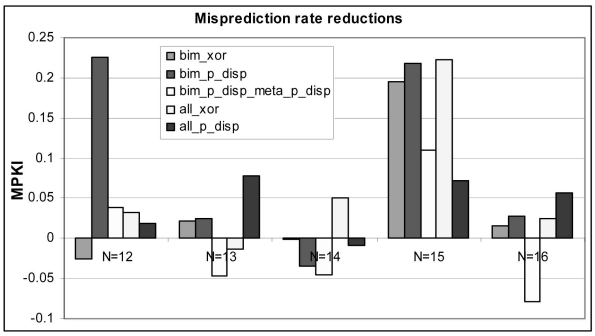

Fig. 8. Miss rate reductions (the higher the better) from various indexing functions on 2bcgskew branch predictors with different table sizes.

hysteresis table (2^{N-2} bits). The baseline 2bcgskew hashing functions use pc and $(N-11)$ GHR bits for the bimodal and metatables, pc and $4*(N-11)$ and $8*(N-11)$ GHR bits for the two g-share tables. As the exhaustive combinations of different indexing functions present a large design space, we focus on the indexing functions with low complexity. For 2bcgskew predictors with N = 12, 13, 14, 15, 16 (i.e., from 4K to 64 K bits), the baseline misprediction rates are 12.90, 9.48, 6.87, 5.73, and 4.88 MPKI, respectively. The reductions achieved from various indexing functions, including the bitwise XOR on the bimodal table (labeled "*bim_xor*"), the prime displacement on the bimodal table (labeled "*bim_p_disp*"), different prime displacement on the bimodal and meta table (labeled "*bim_p_disp_meta_p_disp*"), the bitwise XOR on all tables (labeled "*all_xor*"), and different prime displacement on all tables (labeled "*all_p_disp*"), are shown in Fig. 8.

From Fig. 8, it can be seen that simple indexing functions such as the prime displacement mapping on the bimodal table can significantly reduce the misprediction rates for all 2bcgskew predictors except N = 14, for which the "*all_xor*" mapping produces better improvement. The prime-displacement mapping upon all tables achieves the highest improvement for a 64K-bit 2bcgskew predictor. For a 32K-bit prediction table, all indexing approaches improve the prediction accuracy. Detailed examination upon individual benchmarks reveals that the original 2bcgskew indexing functions have severe pathological behavior with one FP benchmark and our indexing functions effectively avoid it and improve the prediction accuracy significantly on average (e.g., a 3.8 percent improvement from "*bim_p_disp*").

6 Conclusions

In this paper, we explore various indexing functions to reduce conflict aliasing in different branch predictors. The results show that simple indexing functions such as the bitwise XOR mapping and prime-displacement mapping can improve prediction accuracy significantly for perceptron predictors. Slight improvements are also observed for g-share predictors using simple XOR and prime-displacement schemes. Multiple simple indexing functions, in addition, are effective to improve prediction accuracy for 2bcgskew predictors. Considering the negligible cost of these simple indexing functions, we argue that the indexing functions should be carefully explored in designing a highly accurate branch predictor.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable suggestions to improve the paper.

References

[1] F. Bodin and A. Seznec, "Skewed-Associativity Improves Performance and Enhances Predictability," *IEEE Trans. Computers*, vol. 46, 1997.
[2] I.K. Chen, J.T. Coffey, and T.N. Mudge, "Analysis of Branch Prediction via Data Compression," *Proc. Seventh Intl Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, 1996.
[3] J. Frailong, W. Jalby, and J. Lenfant, "XOR Schemes: A Flexible Data Organization in Parallel Memories," *Proc. Int'l Conf. Parallel Processing*, 1985.
[4] A. Gonzalez, M. Valero, N. Topham, and J. Parcerisa, "Eliminating Cache Conflict Misses through XOR-Based Placement Functions," *Proc. Int'l Conf. Supercomputing (ICS-97)*, 1997.
[5] D. Jimenez and C. Lin, "Dynamic Branch Prediction with Perceptrons," *Proc. Seventh Int'l Symp. High Performance Computer Architecture (HPCA-7)*, 2001.
[6] D. Jimenez and C. Lin, "Neural Methods for Dynamic Branch Prediction," *ACM Trans. Computer Systems*, 2002.
[7] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee, "Using Prime Numbers For Cache Indexing to Eliminate Conflict Misses," *Proc. 10th Int'l Symp. High Performance Computer Archictecture (HPCA-10)*, 2004.
[8] S. MacFarling, "Combining Branch Predictors," technical report, DEC, 1993.
[9] P. Michaud, A. Seznec, and R. Uhlig, "Trading Conflict and Capacity Aliasing in Conditional Branch Predictors," *Proc. 24th Int'l Symp. Computer Architecture (ISCA-24)*, 1997.
[10] B. Rau, "Pseudo-Randomly Interleaved Memories," *Proc. 18th Int'l Symp. Computer Architecture (ISCA-18)*, 1991.
[11] S. Sechrest, C. Lee, and T. Mudge, "Correlation and Aliasing in Dynamic Branch Predictors," *Proc. 23rd Int'l Symp. Computer Architecture (ISCA-23)*, 1996.
[12] A. Seznec, "An Optimized 2bcgskew Branch Predictor," technical report, IRISA, 2003.
[13] P. Shivakumar and N. Jouppi, "CACTI 3.0: An Integrated Cache Timing, Power, and Area Model," WRL Technical Report 2001/2, Aug. 2001.
[14] A. Talcott, M. Nemirovsky, and R. Wood, "The Influence of Branch Prediction Table Interference on Branch Prediction Scheme Performance," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques (PACT)*, 1995.
[15] N. Topham and A. Gonzalez, "Randomized Cache Placement for Eliminating Conflicts," *IEEE Trans. Computer*, vol. 48, no. 2, Feb. 1999.
[16] L. Vintan and M. Iridon, "Toward a High Performance Neural Branch Predictor," *Proc. Int'l Joint Conf. Neural Networks*, 1999.
[17] T.-Y. Yeh and Y. Patt, "Alternative Implementations of Two-Level Adaptive Branch Prediction," *Proc. 22nd Int'l Symp. Computer Architecture (ISCA-22)*, 1995.
[18] The First JILP Championship Branch Prediction, http://www.jilp.org/cbp, 2004.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.