A Novel Tag Access Scheme for Low Power L2 Cache

Hyunsun Park, Sungjoo Yoo, Sunggu Lee
Department of Electronic and Electrical Engineering
Pohang University of Science and Technology (POSTECH)
{lena0911, sungjoo.yoo, slee}@postech.ac.kr

ABSTRACT

Tag comparisons occupy a significant portion of cache power consumption in the highly associative cache such as L2 cache. In our work, we propose a novel tag access scheme which applies a partial tag-enhanced Bloom filter to reduce tag comparisons by detecting per-way cache misses. The proposed scheme also classifies cache data into hot and cold data and the tags of hot data are compared earlier than those of cold data exploiting the fact that most of cache hits go to hot data. In addition, the power consumption of each tag comparison can be further reduced by dividing the tag comparison into two micro-steps where a partial tag comparison is performed first and, only if the partial tag comparison gives a partial hit, then the remaining tag bits are compared. We applied the proposed scheme to an L2 cache with 10 programs from SPEC2000 and SPEC2006. Experimental results show average 23.69% and 8.58% reduction in cache energy consumption compared with the conventional serial tag-data access and the other existing methods, respectively.

1. Introduction

As the memory wall problem becomes more significant, the cache tends to occupy more on-chip resource consuming more power. The dynamic power consumption in cache consists of two parts: tag comparison and data access.[1] In the case of highly associative caches, e.g., 16-way L2 cache, where a serial tag-data access is applied, the tag comparison can occupy a significant portion of total power consumption. Assume a 16-way L2 cache with 48b address and 64B lines. The tag size is 4B and the tag comparison requires accessing 4B*16 = 64B data. Thus, the energy consumption in tag comparison is comparable to that in data access.

1.1 Existing Work

There have been several studies on reducing tag comparisons. Existing work can be classified into three categories: way prediction, per-way cache miss prediction and cache decay. In way prediction [1][2][3], the cache way which is most likely to give a hit, mostly the MRU way, is predicted and its tag is first compared. Then, if it gives a hit (called fast hit), then the corresponding data is accessed. If it gives a miss, the remaining tags are compared (for slow hit or miss). In [1], one MRU way is predicted while, in [2][3], multiple MRUs are predicted to give a better accuracy in way prediction. The limitation of way prediction methods is that in case of mis-prediction of MRU way or cache miss, all the remaining tags other than MRU need to be compared for slow hit or miss.

In the second category of predicting per-way cache misses, a non-membership function called Bloom filter is often utilized [4][5][6]. The Bloom filter, utilized on a way basis, can predict cache misses with high probabilities (to be explained in more detail in Section 2). Thus, it can reduce the overhead of tag comparison significantly especially in case of cache misses. However, its accuracy comes at the cost of additional power consumption in accessing the Bloom filter and its area overhead. In [7], a fully associative buffer is managed to hold four tag bits for each cache line and, for predicting cache misses, partial tag comparisons are performed, i.e., the four tag bits are compared with the tag of incoming address. The benefit of the methods in the second category is limited by the accuracy of predicting a per-way cache miss since the tag needs to be compared consuming power if its way's prediction is not successful.

In the third category called cache decay, temporal locality in cache accesses is exploited. Based on the estimation on the liveness of each cache line, the cache is decayed. That is, if a cache line is not expected to be accessed in near future, then it is evicted [8]. The eviction reduces the number of valid tags thereby reducing the power consumption in tag comparison. In [6], cache decay and Bloom filter are combined to further reduce tag comparisons. The limitation of cache decay-based methods is additional energy consumption in main memory due to increased cache misses (i.e., more memory accesses) incurred by the evicted cache lines.

1.2 Our Contribution

Our contribution is summarized as follows.

(1) We present a partial tag-enhanced Bloom filter to improve the accuracy of predicting per-way cache misses. Based on our observation that singleton entries (entries with the counter value of one) dominate non-zero entries in the counting Bloom filter, our scheme manages a part of tag called partial tag in the Bloom filter and significantly increases the accuracy of Bloom filter prediction thereby reducing tag comparisons in case of per-way cache misses.

(2) We present a method of hot/cold tag access with feedback-directed data liveness control. Compared with conventional cache decay which originally targets leakage power reduction in cache, our scheme controls the liveness of cache line in a more fine-grained and feedback-directed manner in order to minimize the power consumption of tag comparisons.

(3) We present a two-level partial/full tag comparison to further reduce the power consumption per tag comparison. This method divides the conventional tag comparison into two micro-steps where a partial tag comparison is first applied. In case of partial tag miss, we can avoid the comparison of remaining tag bits thereby reducing the power consumption of tag comparison.

This paper is organized as follows. Section 2 gives preliminaries and explains our motivation. Section 3 presents our tag access scheme. Section 4 reports experimental results. Section 5 concludes the paper.

[1] In our work, we target low power CMOS process technology and scenarios where the switching power dominates total power consumption. In our experiments, we take into account both switching and leakage power consumption.

978-3-9810801-7-9/DATE11/©2011 EDAA

2. Preliminaries and Motivation

2.1 Bloom Filter for Predicting Cache Misses

Figure 1 illustrates the counting Bloom filter (in short, Bloom filter throughout this paper) used in our work. The Bloom filter (BF) consists of BF entries (counters) and a hash function mapping an input address to one of BF entries. Note that each cache way has its own BF. The figure illustrates how the BF is managed and used to detect the non-existence of an input address, i.e., cache miss for the corresponding way called per-way cache miss. In Figure 1 (a), an address, 0x100 is entered into the BF (i.e., the corresponding cache line is fetched to the cache way) and the counter value of its entry is incremented by one. In Figure 1 (b), another address, 0x300 is entered. It shares a BF entry with the address 0x100. Thus, the counter value is incremented by one and set to two. Note that when an entry exits the BF, i.e., a cache line is evicted from the cache, the counter value of the corresponding BF entry is decremented by one.

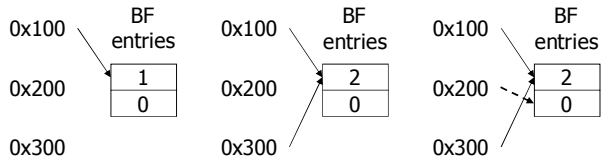

Figure 1 Bloom filter example

In Figure 1 (c), when a new address 0x200 is about to be entered (which corresponds to the case where a new cache access arrives at the cache), the existence of the address 0x200 in the cache way is checked. Since the counter value of corresponding BF entry (shown by the dashed arrow) is zero, we can detect that the address 0x200 is not in the cache way, i.e., a per-way cache miss. In this case, we do not need to perform a tag comparison thereby saving the power consumption for the cache way otherwise required for tag comparison.

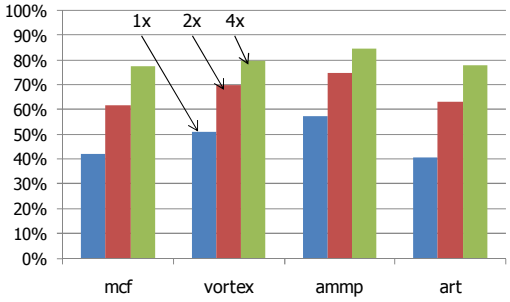

Figure 2 Majority of non-zero counter values are one in the BF

The prediction cannot give false negative, i.e., predicting a cache miss in case of cache hit. However, it can give a false positive, i.e., predicting a cache hit in case of cache miss. Figure 1 (b) illustrates how a false positive can occur. Assume that the request to access address 0x300 just arrives at the cache and its existence is checked with the BF. Since its entry, which is shared by the address 0x100, has a positive counter value of one, we cannot tell that the address 0x300 is not in the cache. Thus, we need to search the cache way, i.e., perform a tag comparison with the address. However, as shown in Figure 1 (a), the address 0x100 set the counter value to one previously. Thus, the address 0x300 is not in the cache way when the address 0x300 arrives at the cache in Figure 1 (b). Such a false positive degrades the BF accuracy significantly and finally requires larger BFs incurring large area and power overhead [5].

In our work, we tackle the problem of false positive in the BF. The main reason of false positive is that the BF cannot tell which address is in the cache when the corresponding entry has a positive counter value. According to our observation, most of positive counter values in the BF are 'one'. Figure 2 (obtained by running our simulations with benchmarks) shows that more than 60% of positive counter values are one in the case of BF where the number of BF entries is 2x that of cache tags. Our work aims at improving the BF accuracy in such a case that the counter value is one. To do that, we manage the partial tag information of cache lines mapped to the BF entry. More details will be given in Section 3.2.

2.2 Hot/Cold Cache Lines and Tag Comparison

Figure 3 illustrates a cache line's life. During the period between t_1 and t_2, it is frequently referenced. Between t_2 and t_3, it is no longer referenced. At time t_3, it is evicted. We call the two periods hot and cold periods, respectively, as shown in the figure. We call cache lines in hot (cold) period hot (cold) cache lines.

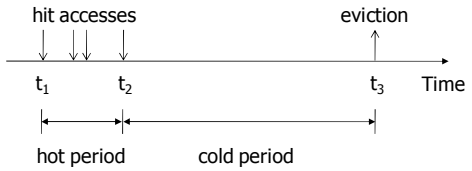

Figure 3 A cache line's life

Hot/cold information of cache lines is utilized in several studies, e.g., deadblock-correlated prefetch[2] [9], virtual victim cache [10], etc. Hot/cold information can be managed in terms of timer [11], counter [12], traces [9], etc. For instance, in the case of counter-based hot/cold management, each cache line is equipped with a liveness counter. Whenever a cache line is referenced, its liveness counter is initialized to a value, TH. The counter is decremented by one every N (= 16 in our experiments) references to the cache. If the counter reaches zero[3], the corresponding cache line is considered to be cold.

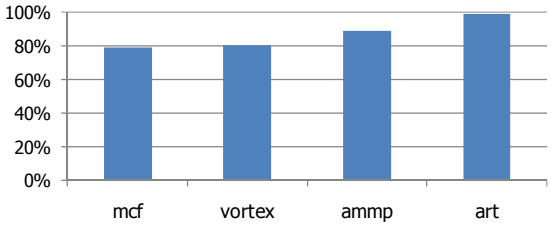

Figure 4 Fraction of hot data in cache hit (at the optimum TH)

In our work, the hot/cold information is exploited in order to reduce tag comparisons. Our method is based on the observation that hot cache lines have higher probabilities of being referenced than cold cache lines. Figure 4 shows that the fraction of hot data in cache hits in our experiments where the counter-based method is used for hot/cold information. As the figure shows, the majority of cache hits go to hot cache lines. Our idea is that on a cache access, hot cache

[2] We use the term 'cold' cache line instead of the term 'deadblock' used in existing work in order to differentiate two liveness management methods, ours and cache decay.

[3] In Section 2.2, we consider the cache line whose liveness counter is zero to be a cold cache line. However, as explained in Section 3.3, for an efficient implementation of feedback-directed control, we consider the tag whose liveness counter is one to be a cold cache line in our implementation.

lines are searched (one cycle) earlier than cold cache lines. In case of hits to hot cache lines, we can save tag comparisons with cold cache lines, which can reduce the average number of tag comparisons. Compared with conventional cache decay-based methods, our difference is that the cold data are not evicted but their tag comparisons are performed later than those of hot data in order to avoid the overhead of additional cache misses caused by decay-based eviction.

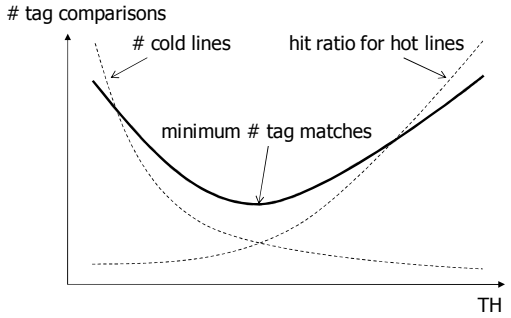

Figure 5 Relationship between # tag comparisons and TH

Figure 5 illustrates that the number of tag comparisons is a function of the initialization value of liveness counter, TH. As shown in the figure, as TH increases, the hit ratio for hot lines increases which is desirable for reducing tag comparisons in our method. However, as TH increases, the number of hot lines (=# ways - # cold lines) also increases. Thus, the efficiency of tag comparison for hot lines decreases since more tag comparisons for hot cache lines will be done in such a case. Thus, as shown in Figure 5, we can obtain a function of number of tag comparisons where we have a minimum level in between the minimum and maximum TH values. If we can set TH to the one giving the minimum number of tag comparisons, we can reduce the power consumption due to tag comparisons. The relationship between TH and the number of tag comparisons (illustrated in Figure 5) changes over different programs and different phases for a program. Thus, we propose a method of determining TH by tracking the dynamically changing behavior of programs (Section 3.3).

3. Proposed Tag Access Scheme

3.1 Solution Overview

We assume a serial tag-data access in the L2 cache. Figure 6 shows the overall flow of the proposed tag access scheme. On arrival of a new access to the cache, the partial tag-enhanced BF is first checked. If the check gives a per-way cache miss as the result, there is no tag access in the cache way. For the cache ways where their BF checks do not give a prediction of per-way cache miss, the tags of hot cache lines are compared with the input tag (Hot check in the figure). We call the hit in hot check *hot hit*. If there is no hot hit, then the tags of cold cache lines are compared with the input tag (Cold check).

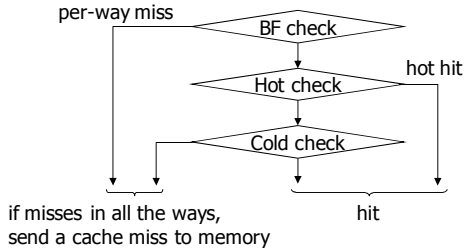

Figure 6 Overall flow of proposed tag access scheme

3.2 Partial Tag-Enhanced Bloom Filter

Figure 7 shows the internal structure of partial tag-enhanced Bloom filter. Each BF entry consists of a tuple <partial tag, S, Z, C>, where S, Z, and C represent singleton, zero, and counter value. If the counter value C is one (zero), flag S (Z) is set to one. As shown in the figure, when a new address arrives at the BF, if the flag Z in the corresponding entry is one (i.e., the counter value is zero), then the BF signals a per-way cache miss. If the counter value is one (i.e., S=1) and the partial tag in the BF does not match with that of the input address, then the BF also signals a per-way cache miss.

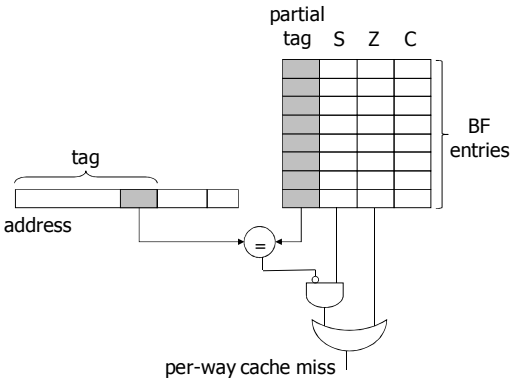

Figure 7 Prediction of cache miss with partial tag-enhanced BF

As shown in the figure, the partial tag in the BF is used only when the flag S = 1, i.e., the cache way has only one address associated with the BF entry. The partial tag is managed as follows. It is first initialized to zero. When an address enters (e.g., data fetch from the memory) and exits (eviction from the cache) the BF, its partial tag is XORed with the existing partial tag of its BF entry as follows.

$$PT_{new} = PT_{cur} \text{ XOR } PT_{enter/exit} \quad (1)$$

where PT_{new} and PT_{cur} are the new and current partial tag and $PT_{enter/exit}$ is the partial tag of entering/exiting address. When the flag S is one, the partial tag contains that of only one address in the cache way. It is because the partial tags of other addresses which previously shared the same BF entry are canceled out with two times of XORing. Assume that an address **a** is associated with BF entry **i**. After the address **a** is evicted from the cache, the partial tag of BF entry **i**, PT_i is calculated as follows.

$$PT_i = PT_{init} \text{ XOR } \ldots \text{ XOR } PT_a \text{ XOR } \ldots \text{ XOR } PT_a \quad (2)$$

There are two occurrences of the partial tag of evicted address **a**, PT_a, the first one for BF entry and the second one for BF exit. As shown in the equation, XORing the same partial tags cancels out each other. Thus, when the flag S of a BF entry is '1', the partial tag in the BF entry contains that of only one address associated with the BF entry. Note that the partial tag management does not incur any new BF access since each address needs to access the BF twice (once for each of BF entry and exit).

3.3 Feedback-directed Hot/Cold Management

The goal of hot/cold line management is to determine TH (a global value for the entire cache) which minimizes the number of tag comparisons as explained in Section 2.2. Figure 8 illustrates how to adjust TH during runtime by tracking the dynamically changing behavior of programs. As shown in the figure, we manage three global counters (for the entire cache) to track the numbers of tag comparisons at the three threshold values of TH_{cur}-d, TH_{cur}, and TH_{cur}+d, where TH_{cur} represents the current threshold value and d is a user defined value for the range of tracking.

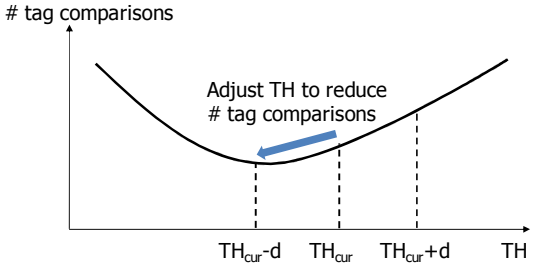

Figure 8 Tracking the minimum number of tag comparisons

The principle of tracking is simple. As shown in Figure 8, we compare the values (i.e., the numbers of tag comparisons accumulated over a time period) of the three counters. Then, we change TH_{cur} to the one whose counter value is the minimum among the three. In Figure 8, we set the TH value to TH_{cur}-d since it gives the lowest number of tag comparisons. The TH value is adjusted periodically (once every 1024 cache accesses in our experiments). On the beginning of a period, the three counters are reset. In our work, we set d to 1. Thus, we explain our method with the three counters for TH_{cur}-1, TH_{cur}, and TH_{cur}+1.

Managing the three global counters requires that each cache line is equipped with three liveness counters (LCs) for the three cases of TH values, which incurs a high area overhead. In our work, we present an efficient implementation which utilizes a single LC per cache line while being able to calculate the three global counters.

```
1  On each cache access,
2  If hot hit
3     Counter(THcur+1) += # tags whose LC >=1
4     Counter(THcur) += # tags whose LC >= 2
5     If LC(addr) == 2
6        Counter(THcur-1) += # ways
7     Else // LC(addr) > 2
8        Counter(THcur-1) += # tags whose LC >=3
9  Else If cold hit
10    If LC(addr) == 1
11       Counter(THcur+1) += # tags whose LC >=1
12    Else // LC(addr) = 0
13       Counter(THcur+1) += # ways
14    Counter(THcur) += # ways
15    Counter(THcur-1) += # ways
16 Else
17    Counter(THcur+1) += # ways
18    Counter(THcur) += # ways
19    Counter(THcur-1) += # ways
```

Figure 9 Counting the numbers of tag comparisons

Figure 9 and 10 show the procedure and an example to calculate the three counters. In Figure 10, we assume a set in the 4-way set associative cache. The four numbers in the rectangles represent LCs for four cache lines in a set. The LCs for the current TH_{cur} are assumed to be 5, 0, 1, and 2. If we had two more LCs for the cache line to track the LCs for TH_{cur}-1 and TH_{cur}+1, we would have a set of numbers (4, 0, 0, 1) for TH_{cur}-1 and another set of numbers (6, 1 or 0, 2, 3) for TH_{cur}+1 as shown in the figure.

Assume there is a hot hit to the fourth cache line as the thick arrow (denoted with '(1) hot hit') shows. Since it is a hot hit, the number of tag comparisons for TH_{cur} will be two, i.e., the number of tags whose LCs >= 2 as shown in Figure 10 and line 4 of Figure 9. Note that, as mentioned in Section 2.2, the cache line with LC <=1 is considered to be cold in our implementation.

If the TH value were TH_{cur}+1, then the number of tag comparisons would be the number of tags whose current LCs >=1 (line 3 in Figure 9), i.e., 3 (W0, W2, and W3 in Figure 10), which gives one more tag comparisons than the case of TH_{cur}. Thus, in this case, TH_{cur} gives a less number of tag comparisons than TH_{cur}+1.

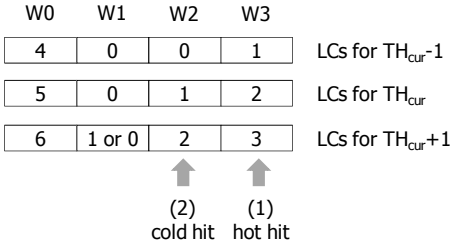

Figure 10 Example of calculating three global counters

If the TH value were TH_{cur}-1, depending on the LC value of currently accessed line, the counter for TH_{cur}-1 is calculated differently. If the LC value of currently accessed line (LC(addr) in Figure 9) is 2 (line 5 in Figure 9 and as shown in Figure 10), TH_{cur}-1 would not give a hot hit, but a cold hit thereby requiring the Cold check which finally requires # ways of tag comparisons. Thus, the counter for TH_{cur}-1 is incremented by # ways in this case (line 6 in Figure 9). However, if LC(addr) were larger than 2 (line 7), though it is not the case in Figure 10, TH_{cur}-1 could further reduce the number of tag comparisons. In this case, the counter for TH_{cur}-1 is incremented by the number of tags whose LC >= 3 (line 8). Thus, it can give a less number of tag comparisons than the case of TH_{cur}. We calculate the counters in a similar way in the cases of cold hit (the arrow denoted with '(2) cold hit' in Figure 10) and miss as shown in Figure 9. After accumulating the three global counters, at the end of period, we choose the TH value whose global counter gives the minimum number of tag comparisons among the three counters. Then, we set the global TH value to that one for the next period. Note that this method requires only three global counters instead of additionally requiring two more LCs per cache line.

3.4 Two-level Partial/Full Tag Comparison

The method of two-level partial/full tag comparison reduces tag comparisons on the bit granularity. Figure 11 shows how to perform a tag comparison in two micro-steps. In the first step, we compare a partial tag (shaded rectangle in the figure) of the input address with those of the tags in the cache (e.g., partial tags of cold cache lines in case of cold check). If the partial tag comparison gives a miss, then there is no need for further comparison of remaining part of the tag for the corresponding cache way. In case of partial hit, the remaining part of tag is compared to give the full comparison result.

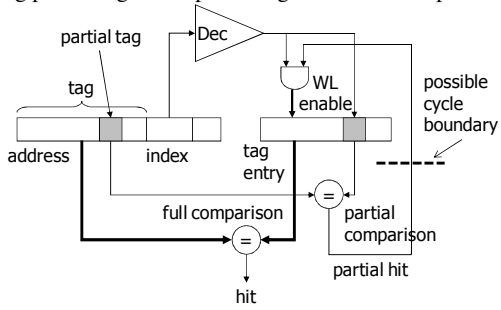

Figure 11 Partial/full tag comparison

The method in Figure 11 can reduce the average number of involved bits in tag comparison. In the tag array, the selection of remaining part of tag is controlled by ANDing the partial hit and the original word line (WL) enable signal as shown in the figure.

The two micro-steps of partial and full tag comparison can be performed in one or two clock cycles depending on the target clock frequency. In the case of high frequency operation, the proposed

method can incur one additional cycle in case of full tag comparison by performing partial and full comparisons in two clock cycles. In our experiments, we apply the partial/full tag comparison only to the cold check and report the impact of additional latency.

4. Experiments

4.1 Experimental Setup

We performed experiments with a cycle-accurate model of target architecture consisting of CPU core, L2 cache, and DRAM memory controller. Table 1 shows the architectural details.

Table 1 Architecture details

Component	Details
CPU core	Tensilica LX2 (7 stage pipeline) [17], 48b address, 64b data, 4-way 16KB I/D, 400MHz
L2 cache	16 and 32-way 256KB, I/D shared, 64B cache line, 400MHz 5 pipe stages: input buffering, tag retrieval, tag comparison, data access, and output
Memory controller	FR-FCFS [13], express path (1+CL latency in case of no previous request), open row scheme
Memory	32b DDR2 800, CL/t_{RP}/t_{RCD}=12.5ns/12.5ns/12.5ns

The L2 cache model was designed in SystemC based on the L2 cache in OpenSparc T2 [14]. Each way in the L2 cache is equipped with a Bloom filter. The size of Bloom filter is varied between 2x and 4x the number of tags in the corresponding way [5].

For the power estimation of L2 cache, we utilize CACTI with 32nm LP technology [15]. In terms of the cycle latency of each step in Figure 6, each of hot check (together with the BF access), cold check, partial and full tag comparisons takes one cycle, respectively. We use DRAMsim [16] for DRAM power estimation in order to evaluate decaying Bloom filter [6] since the method incurs additional energy consumption in DRAM due to increased miss rate while the other methods do not change the number of DRAM accesses. In addition, in order to find the best threshold of cache decay in decaying Bloom filter, we sweep the threshold and utilize the best threshold for the results of decaying Bloom filter. We run 10 programs from SPEC2000 and SPEC2006 on the commercial simulator of Tensilica LX2 [17].

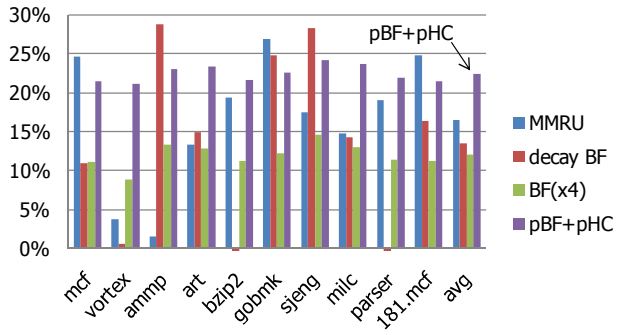

Figure 12 Reduction in energy consumption (16 way)

4.2 Experimental Results

In our experiments, we compare three existing solutions (way prediction [3], Bloom filter only method [5], decaying Bloom filter [6]) and ours in terms of energy consumption and performance overhead. Figure 12 shows the comparison of energy reduction with respect to the baseline of serial tag-data access in case of 16 way cache. Our scheme gives average 22.45% reduction in energy consumption while existing solutions give average 11.98%~16.55% reduction. Note that decaying Bloom filter includes an increase in DRAM energy consumption due to additional misses.

Figure 13 shows a decomposition of the effects of proposed method for four programs in the 32 way cache. Tag comparisons consume more power in the 32 way cache than in the 16 way cache. Thus, the proposed method becomes more effective and gives average 23.69% and 8.58% reduction in energy consumption compared with the baseline and existing methods, respectively. Figure 13 shows that partial tag-enhanced BF (pBF) contributes 21.18% to the energy reduction compared with the baseline. Hot/cold checks (HC) contribute 10.87%. Two-step partial/full tag comparison contributes 12.16% comparing pHC (HC and partial/full tag comparison) and HC.

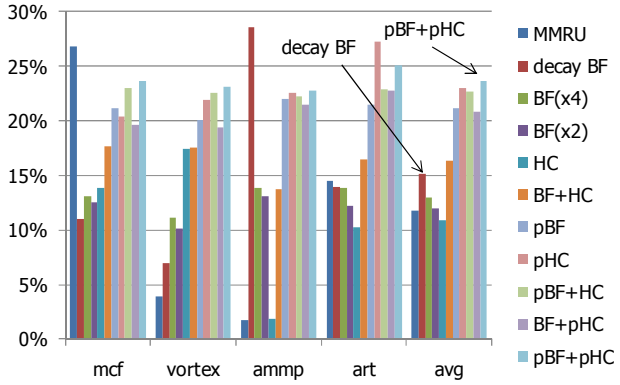

Figure 13 Decomposition of effects of proposed method (32 way)

Figure 14 shows the comparison of energy consumption between the baseline of serial tag-data access (Base_AVG, average of the four programs) and ours in the 16-way cache case. The figure shows that the proposed scheme reduces the energy consumption of tag comparison by 87%~91% while incurring the overhead of Bloom filter thereby giving 21%~23% reduction in total energy consumption.

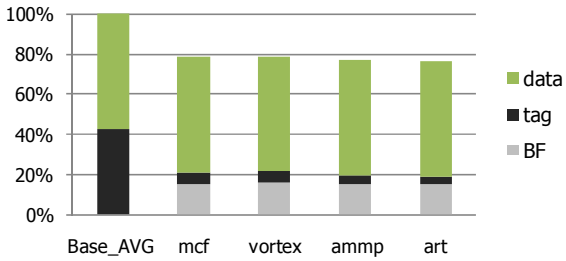

Figure 14 Energy decomposition: serial tag-data vs. ours

Figure 15 compares the number (normalized to the serial tag-data access case) of tag comparisons between BF only method (BF) [5] and the proposed partial tag-enhanced BF (pBF). As the figure shows, the proposed pBF with 2x size gives much less number of tag comparisons, i.e., much higher prediction accuracy than the original 2x and 4x BFs. In terms of area, both our 2x pBF (e.g., 8.45% in 512KB cache) and the original 4x BF (8.43%) are almost the same size. However, our pBF consumes by 20.15% less energy per BF access than the original 4x BF. The combination of less power consumption and more accurate prediction (shown in Figure 15) gives the gain in energy consumption in Figures 12 and 13 compared with the original BF-based method.

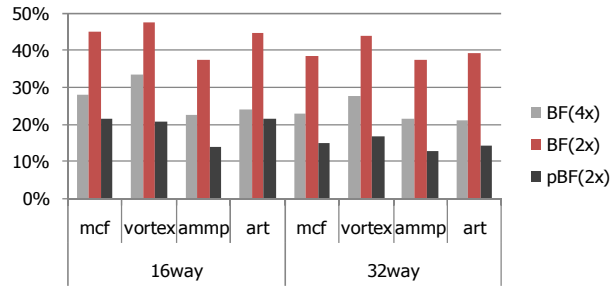

Figure 15 Number of tag comparisons

Figure 16 shows the decomposition of hit/miss ratio. The sum of hot and cold hits corresponds to hit ratio. Full-way miss represents the case that BF prediction at all the cache ways gives the prediction of miss. In such a case, no tag comparison is performed in the entire cache. Figure 16 shows that, in case of high hit (miss) ratio, hot hit (cold-partial miss) occupies the majority of cache hits (misses).

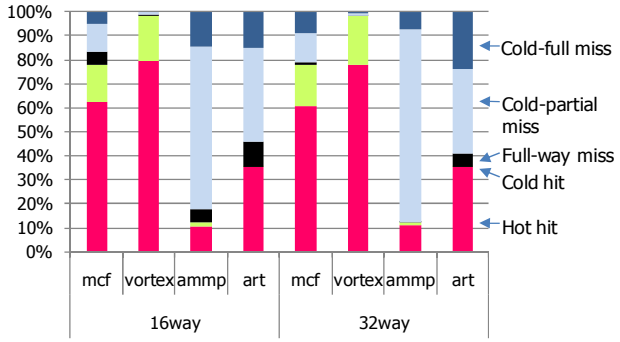

Figure 16 Decomposition of hit and miss ratio

Figure 17 shows the comparison of runtime overhead in terms of total cycles for the 16 way cache. The proposed scheme gives 0.57% overhead in total cycles. It is because the proposed tag access scheme can require additional cycles in cold check and partial tag comparison. Note that the methods (BF and pBF) based only on Bloom filter do not incur performance degradation since the Bloom filter operation can be done in tag comparison stage [5].

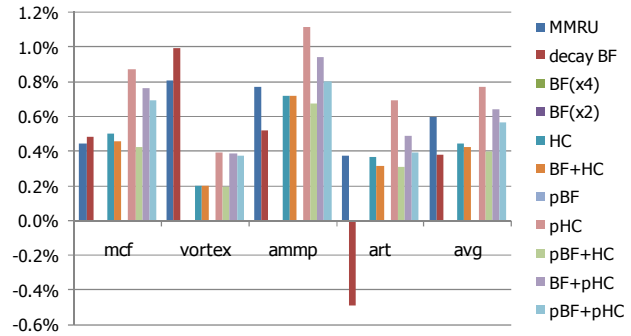

Figure 17 Comparison of runtime overhead in total cycles

Figure 18 shows a snapshot of TH tracking in mcf. As the figure shows, TH varies over time. The figure shows that the proposed feedback-directed method (solid line) tracks well the varying ideal TH (dashed line) which is obtained, in an off-line calculation, as the TH value which gives the minimum tag comparisons.

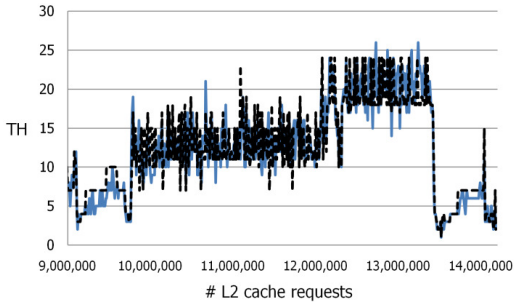

Figure 18 TH tracking (mcf case)

5. Conclusion

In this paper, we presented a novel tag access scheme for low power L2 cache. First, a partial tag-enhanced Bloom filter is proposed to improve the prediction accuracy of per-way cache misses. Second, a feedback-directed method of hot/cold tag access exploits temporal locality to reduce tag comparisons. Third, a two-level partial/full tag comparison reduces the bit-level tag comparisons by avoiding full tag comparison in case of miss in partial tag comparison. The presented scheme gives 23.69% and 8.58% reduction in the total energy consumption of L2 cache compared with the conventional serial tag-data access and other existing solutions, respectively.

6. Acknowledgement

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2010-0007909 and 2010-0015336) and by Samsung Electronics and IC Design Education Center (IDEC).

7. References

[1] K. Inoue et al., "Way-Predicting Set-Associative Cache for High Performance and Low Energy Consumption," Proc. ISLPED, 2009.
[2] M. Powell et al., "Reducing Set-Associative Cache Energy via Way-Prediction and Selective Direct-Mapping," Proc. MICRO, 2001.
[3] Z. Zhu and X. Zhang, "Access-Mode Predictions for Low-Power Cache Design," IEEE Micro, 22 (2), 2002.
[4] J. J. Peir et al., "Bloom Filtering Cache Misses for Accurate Data Speculation and Prefetching," Proc. Supercomputing, 2002.
[5] M. Ghosh et al., "Way Guard: A Segmented Counting Bloom Filter Approach to Reducing Energy for Set-Associative Caches," Proc. ISLPED, 2009.
[6] G. Keramidas et al., "Applying Decay to Reduce Dynamic Power in Set-Associative Caches," Proc. HiPEAC, 2007.
[7] C. Zhang et al., A Way-Halting Cache For Low-Energy High-Performance Systems, ACM TACO, 2 (1), pp. 34-54, 2005.
[8] S. Jaxiras et al., "Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power," Proc. ISCA, 2001.
[9] A. Lai et al., "Dead-Block Prediction & Dead-Block Correlating Prefetchers," Proc. ISCA, 2001.
[10] S. Khan et al., "Using Dead Blocks as a Virtual Victim Cache," Proc. ASPLOS, 2009.
[11] Z. hu et al., Timekeeping in the Memory System: Predicting and Optimizing Memory Behavior," Proc. ISCA, 2002.
[12] F. Roesner et al., "Counting Dependence Predictors," Proc. ISCA, 2008.
[13] S. Rixner et al., "Memory Access Scheduling," Proc. ISCA, 2000.
[14] Sun Microsystems, Inc., OpenSPARC™ T2 System-On-Chip (SoC) Microarchitecture Specification, No. 820-2620-10, May 2008.
[15] HP Labs., CACTI 6.5, http://www.hpl.hp.com/research/cacti/.
[16] D. Wang et al., "DRAMsim: A Memory System Simulator," ACM SIGARCH Computer Architecture News, 33 (4), Sept. 2005.
[17] Tensilica, Inc., XPRES Compiler, http://www.tensilica.com/.