Way Guard: A Segmented Counting Bloom Filter Approach to Reducing Energy for Set-Associative Caches

Mrinmoy Ghosh† Emre Özer‡ Simon Ford‡ Stuart Biles‡ Hsien-Hsin S. Lee⋆

†ARM Inc., Austin, TX ‡ARM Ltd., Cambridge, UK ⋆Georgia Tech, Atlanta, GA

ABSTRACT

The design trend of caches in modern processors continues to increase their capacity with higher associativity to cope with large data footprint and take advantage of feature size shrink, which, unfortunately, also leads to higher energy consumption. This paper presents a technique using segmented counting Bloom filters called "Way Guard" to reduce the number of redundant way lookups in large set-associative caches to achieve dynamic energy savings. Our Way Guard mechanism only looks up an average of 25-30% of the cache ways and saved up to 65% of the L2 energy and up to 70% of the L1 cache energy.

Categories and Subject Descriptors

C.1.0 [Processor Architecture]: General; B.3.2 [Memory Structures]: Design Styles– *Cache memories*

General Terms

Design, Experimentation

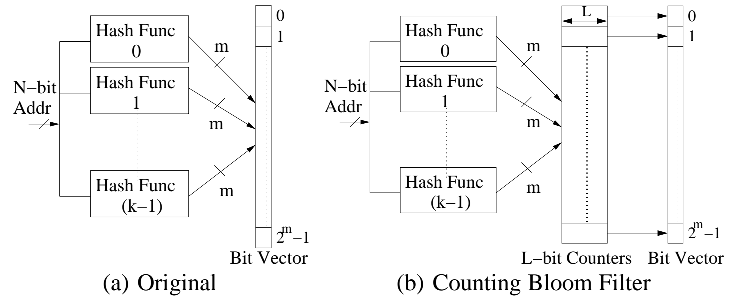
1. INTRODUCTION

Cache hierarchy has become a main consumer of both static and dynamic energy in processors. Even so, the trend in modern processor designs continues to increase both capacity and associativity to accommodate the ever-growing workloads and alleviate conflict misses. For processors employing highly associative caches, the energy consumption gets even worse as N-tag comparisons are needed for each parallel lookup of an N-way cache. In fact, most of the energy consumed for such lookups is redundant as the requested data can only be present in one particular way. This redundancy provides a good opportunity for saving dynamic energy.

In this paper, we propose a technique called *Way Guard* based on segmented counting Bloom filters to exploit these energy saving opportunities. Our scheme uses counting Bloom filters to efficiently skip the lookup of cache lines that do not contain the requested data to save significant energy in cache accesses. Bloom filters are simple, fast structures that can eliminate the need of performing associative lookup especially when the lookup address space is huge. They can replace the expensive set-associative tag matching with a simple bit vector that precisely identifies addresses that have not been observed before. This mechanism provides early detection of events to avoid an associative buffer lookup. This improves energy consumption significantly without adversely affecting performance given the efficient hardware structures.

The rest of this paper is arranged as follows. Section 2 explains Bloom filters. Section 3 describes our segmented Bloom filter and its energy-saving feature. Section 4 explains Way Guard technique. Section 5 describes our simulation methodology and analysis. Section 6 reviews prior techniques. Section 7 concludes.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISLPED'09, August 19–21, 2009, San Francisco, California, USA.
Copyright 2009 ACM 978-1-60558-684-7/09/08 ...$10.00.



(a) Original (b) Counting Bloom Filter

Figure 1: Bloom Filters

2. BLOOM FILTERS

The original Bloom filter concept is depicted in Figure 1(a). It consists of several hash functions and a bit vector. A given *N*-bit address is hashed into *k* hash values using *k* different random hash functions. The output of each hash function is an *m*-bit index that addresses the 2^m entry bit vector, where *m* is much smaller than *N*.

Initially, the Bloom filter bit vector is zero. Whenever an *N*-bit address is observed, it is hashed to the bit vector and the bit value hashed by each *m*-bit index is set to one. When a query is to be made, the given *N*-bit address is hashed using the same hash functions and the bit values are read from the locations indexed by the *m*-bit hash value. If at least one of the bits is 0, it indicates that this address was definitely not observed before. This is called a *true miss*. Whereas, if all of the bit values are 1, the address may have been observed but with no guarantee, which is called a *false hit*.

As the number of hash functions increases, the Bloom filter bit vector is polluted faster. On the other hand, the probability of finding a zero during a query is increased if more hash functions are used. The major drawback of the original Bloom filter is the high false hit rate as it can be quickly filled up with all 1's. Also, once a bit is set, there is no way to reset it. Thus, as more bits are set, the number of false hits increases. To address this issue, the counting Bloom filter [7] shown in Figure 1(b) was proposed for web cache sharing to provide capability of resetting entries in the filter. First, an array of counters is added along with the bit vector of the original Bloom Filter. Each L-bit counter has a one-to-one association with each bit in the bit vector. Queries to a counting Bloom filter is similar with a slight modification: when an address is entered,

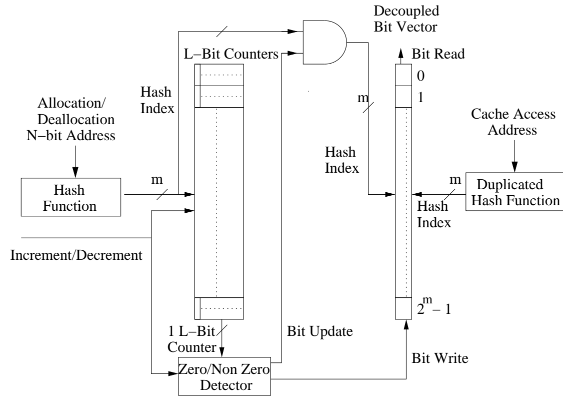


Figure 2: Segmented Bloom filter

each *m*-bit hash index will increment its corresponding counter of the counter array in addition to setting the bit vector. Similarly, when an address is removed from the Bloom filter, each *m*-bit hash index will decrement its corresponding counter. If more than one hash indexes to the same location for a given address, the counter is incremented or decremented only once. Finally, when a counter is reduced to zero, its associated bit in the bit vector will be cleared.

3. SEGMENTED COUNTING BLOOM FILTER

One application of the counting Bloom filter is to keep track of the line-fills and replacements of a cache and indicate whether an address is present in the cache. Query to a counting Bloom filter consumes less energy and quicker than accessing the entire cache. Ghosh *et al.* has shown a cache miss detection technique using a segmented counting Bloom filter [8]. Their design redrawn in Figure 2 contains the counter array (*L* bits per counter) decoupled from the bit vector with a duplicated hash function on the bit vector side. The cache line fill/eviction addresses are sent to the counter array using one hash function while the cache request address from the processor is sent to the bit vector using a copy of the same hash function. The segmented Bloom filter design allows the counter array and bit vector to operate in separate physical locations.

There are several reasons for a segmented Bloom filter: 1) We only need the bit vector, which is smaller than the counter, to obtain the outcome of a query. Decoupling the bit vector enables faster and lower energy accesses to the Bloom Filter. Hence the result of a query issued from the core can be obtained by just looking up the bit vector; 2) The update to the counters is not time-critical with respect to the core. So, the segmented design allows the counter array to run at lower frequency than the bit vector. The vector part being smaller provides fast access time, whereas the larger counter part runs at a lower frequency to save energy. The only additional overhead of the segmented design is the duplication of the hash function hardware. We now describe an innovative application of the segmented counting Bloom filter to avoid unnecessary cache way lookups.

4. WAY GUARD MECHANISM

In this section, we describe a novel application of counting Bloom filters to set-associative caches to determine data presence and save lookup energy. Figure 3 illustrates the design of a 4-way cache with our proposed *Way Guard* mechanism. As shown, each Way Guard, structurally the same as the counting Bloom Filter, consists of a Bit Vector (shown as BV) and an array of counters. Each way of a set-associative cache is assigned one Way Guard. The purpose and functionality of these filters are similar to the Segmented Bloom Filters explained earlier. The only difference is that each Way Guard keeps line-fill and replacement information of the cache way it is guarding. The following two properties are important in understanding how the Way Guard technique works:

1. If the filter indicates that the address is not present in the way it is guarding, then the data is certainly not present in that particular cache way.

2. If the filter indicates that the address is present, then the data may be present in the given cache way.

As such, the filter provides a completely safe indication about the absence of data in the cache way it is guarding. Also, this indication can be performed within a fixed access time, as opposed to previously proposed prediction techniques that contain undesirable variable access times. Figure 3 shows a scenario of an access to a *Way Guard cache*. In the example, since the Way Guard filters of Way 0 and Way 3 indicate a possible hit, only their Tag RAMs need to be checked, which enables the dynamic energy reduction. Even though the scheme incurs extra hardware, the Way Guard only comprises of a bit-vector and counters. Querying a Way Guard only involves checking the bit vector and this consumes much less energy than looking for the address in the Tag RAM it is guarding. Note that, since the filter must be checked prior to the Tag RAMs being selected, there is a potential performance penalty. However, since the filter is fast, the filter access and the Tag RAM selection process can be potentially contained within a cycle. In the worst case, the filter would add one extra cycle to the cache access time.

One implementation variant of the *Way Guard* technique is illustrated by the dotted lines in Figure 3. There are *"n"* Guards each guarding a way of an n-way cache. However, note that each filter is indexed by the same hash function. Given an address, all the filters will check the same index for the presence of the data. Since for all cache accesses all the filters are queried, we propose the following design alternative. In this variant, the vector part of the segmented Bloom filters are coalesced to form the *Way Guard Matrix* shown by the dotted line table. Each row in the matrix contains a bit for each guard filter. This bit will be the bit in the bit vector corresponding to the index of the row. Thus Matrix[i][k] will consist of the ith row of the bit vector of the Way Guard guarding the kth way. During a cache access, this matrix is first queried as shown by the dotted arrows coming from the address and the row of bits obtained for the given index of the address is used to enable only the cache ways that may contain the data. This technique can further reduce energy by using the lower energy matrix structure to filter out unneeded cache lookups going to the Tag RAMs. The bit vectors which consist of a very large number of one-bit entries[1], and the majority of the access cost to the bit vector structure goes in decoding the index. Using one matrix instead of *n* one bit arrays saved *(n-1)* decoder access costs for every access to the Way Guard.

One notable point about our technique is that it can be extended to other implementations of highly associative caches such as CAM tag caches [23]. This can be done by simply adding an AND gate in the path of every CAM comparator. One input to these AND gates is obtained from the Way Guard result to effectively reduce tag comparisons. The detailed evaluation is outside the scope of this paper.

5. EXPERIMENTAL ANALYSIS

5.1 Experimental Framework and Benchmark

We use a modified version of Bochs [10], a full system x86 emulator, to evaluate the energy savings by taking all effects including

[1]The number of entries we chose is four times of the number of cache lines. For a 256KB cache with 32-byte lines, the number of entries in the array is 32768.

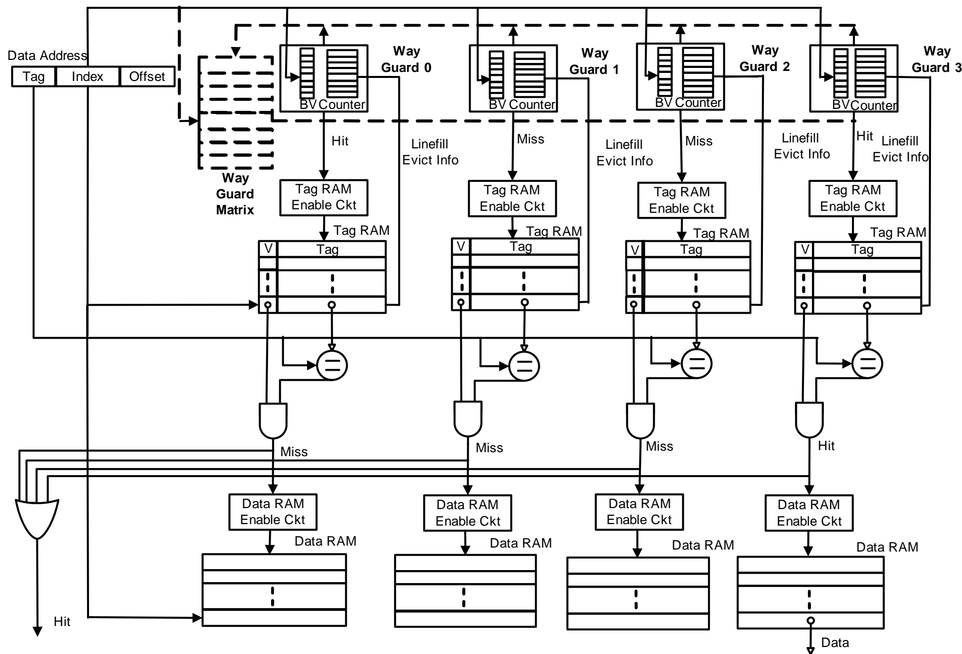


Figure 3: Way Guard Mechanism — Filtering Out Unnecessary Cache Way Lookup

the OS into account. The simulations involved running common Windows applications like Visual Studio and a set of seven SPEC benchmarks running on Windows NT on the Bochs emulator.

To collect cache statistics, we integrate a cache simulator enhanced with our proposed technique into Bochs, which allows us to gather cache statistics of various applications directly running on top of a full OS. Using this framework we can also simulate several different memory hierarchies simultaneously.

The evaluation for *Way Guard* is performed in two stages. In the first stage, only the L2 cache is guarded by the Way Guard filters. So, we used a fixed size of a 2-way 16KB L1 cache, and 30 different configurations of the L2. We varied the capacity by gradually doubling its size from 64KB up to 2MB. The associativity was varied in the same manner from 2 to 32 ways.

The size of each Way Guard filter was chosen to be four times the number of lines of each cache way. We performed experiments with different filter entries and found that having four times the number of entries gave the best trade-off between filter performance and energy savings. The area overhead for all Way Guards in the L2 is 6% of the L2 size. Notice that this 6% relative overhead is irrespective of the cache size, as we always use the heuristic of choosing the Way Guard filter with entries that is four times the number of entries of the number of cache lines it is guarding.[2]

We use three bits per counter in the Way Guard counter array. Since the number of filter entries is four times the number of lines, it is not expected that for a hash function that can evenly distribute entries, a three bit counter will overflow. We observed that for all the experiments performed we did not have a single occurrence of a counter overflow. We implemented a policy that in the unlikely event of a counter overflow, we will stop updating entries for that particular counter and conservatively indicate a possible hit, every time a query indexes to that particular counter entry.

We show energy savings results for both the serial access and parallel access versions of the L2 cache. In a serial access, cache data access follows the tag access only when there is a tag match. In contrast, for a parallel access cache, the data and tag of all ways are enabled in parallel, and the correct data is "muxed" out.

[2]There is also a small overhead of a few logic gates per Way Guard, but the overhead is negligible compared to the size of the filter.

In the second stage, only L1 caches are guarded by the Way Guard. So, we used a 4-way 128KB fixed sized L2 cache, and 20 L1 configurations. Similarly, the L1 capacity was varied from 8KB up to 64KB and their associativity from 2 to 32 ways. The 32-way L1 is similar to what is employed in XScale processors. We assume a parallel access L1 cache for all our experiments. We use the a set of seven SPEC benchmarks that were known to stress the L2 cache. In addition, we also used five MS Windows applications used in desktop systems including the booting of Windows NT, Visual Studio compiling the Bochs source code, an MPEG decoder, a MP3 decoder, and a simple web browsing application. All the above applications show sufficient amount of memory activity to properly illustrate our results. Also, the MS Windows benchmarks help us understand how the Way Guard technique will behave in a real multiprocessing environment.

5.2 Energy Modeling

The L1 and L2 caches, the bit vector, and the counter array were designed using the *Artisan* 90nm SRAM library in order to get an estimate for the dynamic and static energy consumption of the caches and the segmented Bloom filter. The Artisan SRAM generator is capable of generating synthesizable Verilog code for SRAMs in 90nm technology. The generated data-sheet provides the read and write current of the generated SRAM. This gives us an estimate of the dynamic energy per access of such a structure. The data-sheet also provides a standby current from which we can calculate the leakage energy per cycle of the SRAM.

5.3 Energy Savings for Way Guard

To illustrate the effectiveness of the Way Guard, we show the average number of ways looked up (for all the benchmark programs) for hits and misses for all the 30 L2 cache configurations in Figure 4 and Figure 5. A notable observation of these results is that the Way Guard does a very good job in filtering out ways where the data is not present. In a typical case of a cache hit for an 8-way cache, only 2.77 ways need to be looked up for a data access. The average number of ways needed to determine a cache miss is significantly lower than that for hits. To determine a miss, the Way Guard cache checks less than 25% of the ways. Another interesting trend is that the performance of the Way Guards continues to

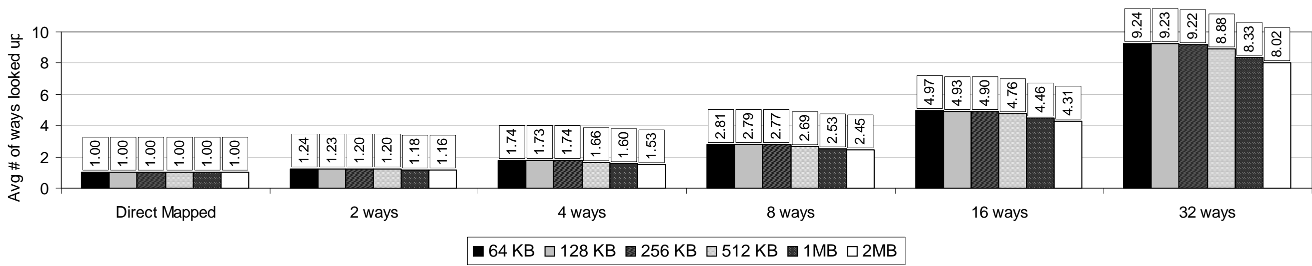


Figure 4: Average Number of Ways Looked Up for Hits in an L2 Cache

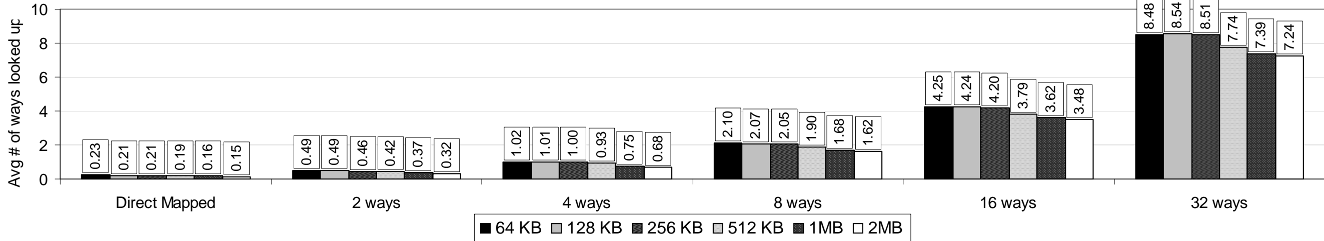


Figure 5: Average Number of Ways Looked Up for Misses in an L2 Cache

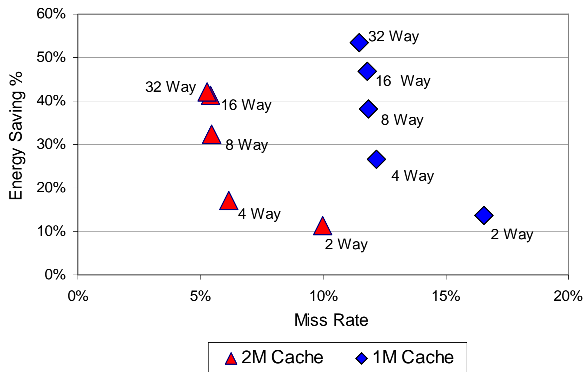


Figure 6: Energy Savings with respect to Miss Rate

improve with increasing cache sizes. This has to do with the sensitivity of the counting Bloom filter performance with its size. Our experiments showed that a larger counting Bloom filter always performs better than a smaller one, even though the size of the Bloom filter is always chosen to be four times the number of cache lines in each way. For a given associativity, since larger caches have larger Way Guards guarding their ways, the performance of the filters in larger caches will be better.

In Figure 6 we try to find out how the total energy savings is affected by the miss rate. The energy savings take into account both dynamic and leakage energy consumptions of the cache as well as those consumed by the Way Guards. The baseline is a normal L2 cache without the Way Guard mechanism. The figure shows the savings obtained for two cache sizes (1MB and 2MB) for the "bzip2" from SPEC benchmark. "Bzip2" is chosen for this illustration as a typical benchmark showing trends reflected in all other benchmarks. We find that the miss rate for a fixed cache size is almost the same for associativities greater than 2. We observe that we get significant savings for up to 53% for a 32-way 2 MB cache. As expected for all cache sizes, the savings increase as the associativity increases. In other words, the effectiveness of the Way Guard is increased with higher associativity. The reason behind this is that the Bloom filters are very effective at indicating absence of data, and can predict more than 80% of cache misses [8]. In a set associative cache lookup, most of the accesses to ways result in misses, that counting Bloom filters are usually good in predicting. In the case of a cache hit only one way has the required data and in the case of a cache miss none of the ways have the data. Thus, a higher associativity gives a greater chance for indicating absence, leading to larger energy savings.

By fixing the associativity, the energy savings decrease with increased cache sizes. One reason for this is that for larger cache sizes the overhead of the Bloom filter also increases. Also larger caches contain lower miss rates. So the relative benefits do not completely account for the larger overheads.

Another trend that was observed was that the relative energy overhead of Way Guard increases with associativity and decreases with cache size. We found that for a small 64KB L2 cache the relative energy overhead of Way Guard ranges from 16% for a 2-way cache to 22% for a 32-way cache. Instead, for a 2MB cache, the overhead is only 5% for 2-way to 14% for 32-way. There are two reasons why the Way Guard's overhead increases with associativity. First, as the associativity increases, more bits (one bit for each way) have to accessed for each access to the cache. Second, as the associativity increases, the performance of Way Guard also improves leading to less total energy consumption. The reason for the relative overhead decreasing with increasing cache size is that, as the cache size increases, the relative energy in accessing the cache becomes relatively larger than that of accessing the Way Guard.

We compared our technique with the Way Halting technique described in [22]. Way Halting uses a fully associative buffer to hold four tag bits for each line of the cache. When the cache is looked up, the way halting buffer matches the stored bits for each way of the corresponding set with the least significant tag bits of the address looked up. If these bits do not match for a particular way, then the lookup will surely miss that way, and the tag comparison for that way is halted, resulting in energy savings. We implemented the way halting scheme in our Bochs infrastructure. We also modeled the power overheads for the way halting technique using the Artisan SRAM generator.

The L2 cache energy savings comparing Way Halting and our Way Guard techniques are shown in Figure 7. The baseline cache for these relative energy numbers is a serial lookup cache. In a serial lookup cache, to reduce the lookup energy consumption, tag comparisons and the retrieval of the data portion of a cache line are done serially, similar to what was described in Figure 3. An access involves two steps starting with a tag comparison. Only if there is a tag match will the corresponding data row be accessed to supply the data line. All benchmark programs show similar energy saving trends in the serial lookup cache. Thus we report the geometric

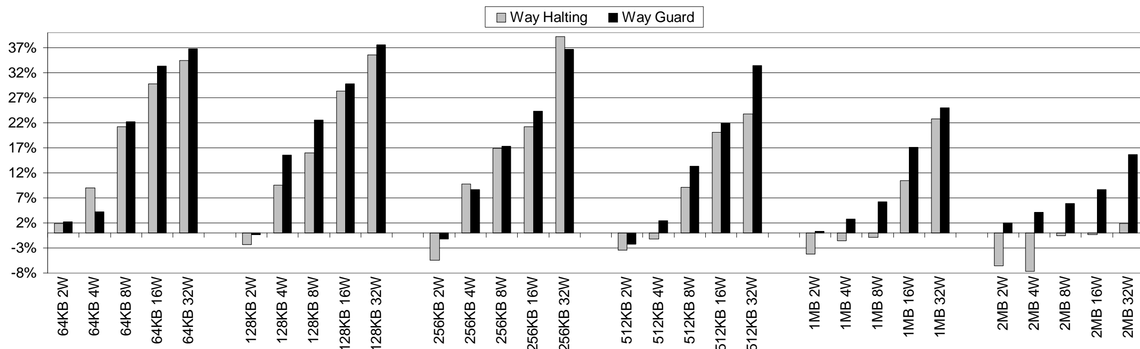


Figure 7: Comparing Way Halting and Way Guard Energy Savings in a Serial Lookup Cache

means of energy savings for all the benchmarks in Figure 7. We see that since the data of all the ways are not fired up, the primary savings with the Way Guard lie in the tag comparisons. We find that for all cache sizes considered, the Way Guard technique is not very effective for caches whose associativity is less than 4. The reason behind this is, for a 2-way cache, Bloom filters save at most one tag comparison for a hit and 2 tag comparisons for a miss. This benefit in most cases does not surpass the extra energy cost needed for checking the Bloom filters for each L2 access. In contrast, it shows energy savings of up to 37% for larger associativity caches. Compared to the Way Halting scheme , the Way Guard technique shows much better energy savings for 27 of the 30 cache configurations. In a typical case a 1MB 16-way L2 has a 17% energy savings for a Way Guard while the Way Halting scheme only achieves 6.3% savings. Also for low associativities, the Way Halting technique does not have any energy savings. For a 2MB 4-way cache, Way Halting technique results in a 7% energy loss. The reason behind this is the high overhead of Way Halting for every cache access, that involves comparing four tag bits for every cache way. In contrast, the Way Guard technique only involves reading “n” bits from a bit vector array, where “n” is the associativity.

We also compared the energy savings of our technique against Way Halting based on a parallel lookup cache. The results in Figure 8 assume that the set-associative cache accesses the same data row for all cache ways in parallel. It can be easily seen that the Way Guard technique performs much better than the Way Halting technique for 25 of the 30 cache configurations. In a typical case, for a 1 MB 4-way cache, the Way Guard cache shows a saving of 32%, while Way Halting manages to improve the energy by 21%. As expected, the results for a parallel lookup cache show similar trends as the results for a serial lookup cache in terms of sensitivity to cache associativity and cache sizes.

For the L1 cache experiment, we consider the L1 to be a high performance parallel access cache, where data and tag are accessed together to achieving a fast hit latency. We applied our Way Guard technique to both the instruction and data caches. For all our experiments we assume a 2-cycle cache access. We assume that our Way Guard lookup can be fit into these two cycles to not affect the L1 cache performance. This assumption is validated by considering the access times of a typical cache configuration (64KB 4-way) and its corresponding Way Guard Bloom filter (2048 entries each way). Using Artisan the access latencies were found to be 0.74ns for the cache and 0.66ns for the Way Guard. The combined access time fits into 2 cycles of an embedded processor like AMD Geode NX 1750 running at 1.4GHz. Note that a normal cache access to this processor will also take 2 cycles, as the 0.74ns access time to the cache is larger than the 0.714ns cycle time.

We first show the geometric means of the normalized energy savings in the L1 I-cache across all the benchmark in Figure 9. In these experiments, our Way Guard technique shows huge benefits up to 68% for a 32-way cache and more than 50% for a 4-way cache. L1

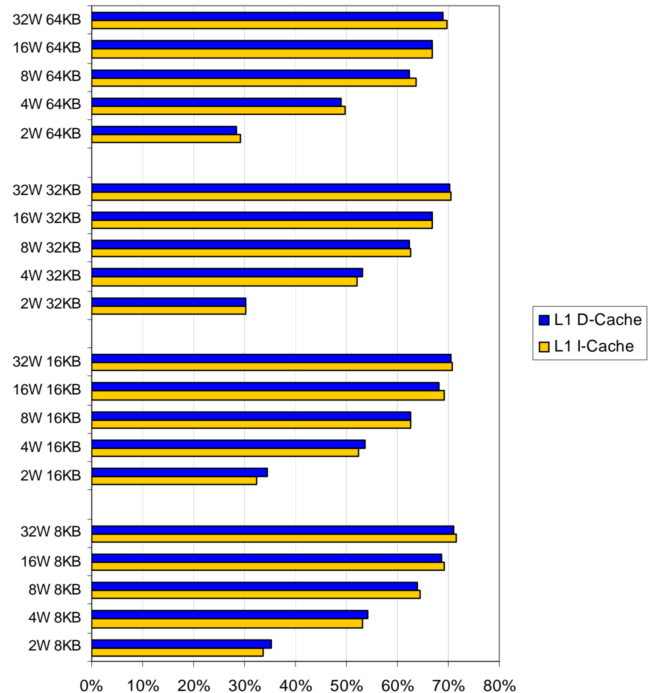


Figure 9: Average L1 I- and D-Cache Energy Savings

caches show huge benefits because it is almost accessed every cycle during execution. For every access, with the help of Way Guard filters, only 25 to 30% of the ways need to be checked.

We also performed similar experiments for the L1 D-cache. The results are also shown in Figure 9. Similar to the I-cache results, the savings obtained using Way Guard despite the overheads are very impressive. In a typical case of a 32KB 4-way L1 cache, a 52% overall energy saving was shown.

6. RELATED WORK

The initial purpose of Bloom filters was to build memory efficient database applications. Since then, Bloom filters have found numerous applications in networking and database areas [2, 6, 13, 5]. They were also applied to microarchitectural blocks for predicting load/store collision in LSQ [20] or optimizing the frequency of load re-execution [19].

The earliest work of tracking cache misses with a counting Bloom filter is given in [17]. A hardware structure called *Jetty* was proposed to filter out L2 cache snoops in SMPs. A *Jetty*-like filter is also used by *Peir et al.* [18] for detecting load misses early in the pipeline so as to initiate speculative execution. Similarly, *Mehta et al.* [15] also uses a *Jetty*-like filter to detect L2 misses early to

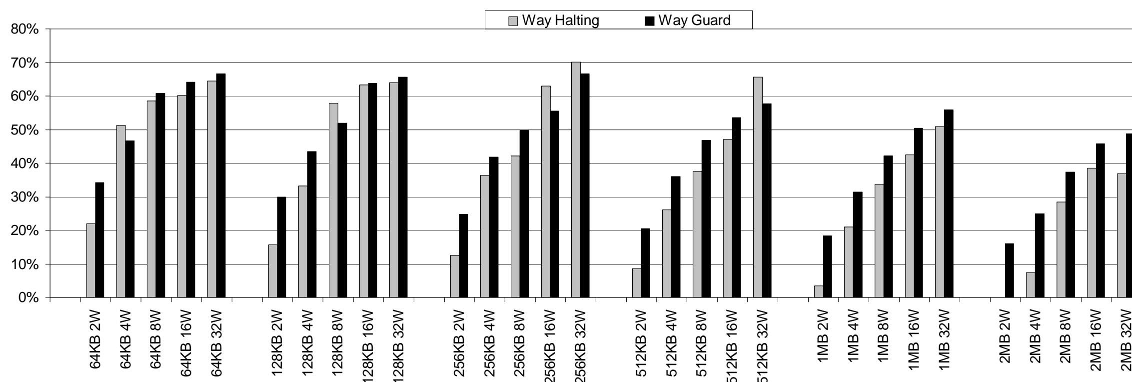


Figure 8: Comparing Way Halting and Way Guard Energy Savings in a Parallel Lookup Cache

stall the instruction fetch for saving energy. *Memik et al.* [16] proposed early cache miss detection hardware encapsulated as *Mostly No Machine (MNM)*. Their goal is to reduce dynamic cache energy and to improve performance by bypassing the caches that will miss. Counting Bloom filters were used in [21] to detect the absence of synonyms for reducing energy consumption in virtually indexed caches. We, on the other hand, propose a decoupled Bloom filter structure where the small bit vector can potentially be kept within the processor core to perform system dynamic and static energy conservation of L1 and L2 caches and the core itself.

The most common way prediction mechanism is to predict the MRU way as proposed in [3]. Similar way prediction techniques were studied in [9, 12]. Another PC-based way prediction scheme was proposed in [1]. However, way prediction has the disadvantage of a large performance and energy loss if the prediction is wrong. One alternative to way prediction is way memoization [14]. Way memoization keeps way information in the I-cache with valid bits that ensure that the way information is correct. However, this technique can only be used in I-caches. Their shortcoming can be overcome by our Way Guard.

Instead of predicting one single way, way estimation techniques were proposed to predict a set of ways where the data is guaranteed to be present for a cache hit. Therefore, way estimation techniques do not incur a large performance loss for a wrong estimation, because a wrong estimate only results in a lookup in the cache when it is missing the cache. One way estimation technique is the sentry tag [4]. Way Halting [22] is an extension of sentry tags. We did comparisons of this with our Way Guard scheme and showed much more energy savings in our method than the sentry tags. Another way estimation technique similar to Way Guard was proposed in [11]. In which the authors tried to predict lines that have decayed. Since this technique incorporates cache decay, it is not suitable for the L1 caches as it may increase the miss rate considerably. Our technique does not increase the miss rate and thus has no appreciable effect on the performance of the system.

7. CONCLUSION

As future applications demand more memory and shrinking feature sizes allow more one-die transistors, processors are inclined to incorporate larger caches with higher associativity. These larger structures, unfortunately, also lead to higher energy consumption. This paper presents an efficient use of the counting segmented Bloom Filter called *Way Guard* to reduce significant dynamic cache energy by filtering out unnecessary way lookup in a set-associative cache. We showed that our technique can be efficiently applied to all levels of the cache hierarchy, obtaining substantial energy savings of up to 70% in both instruction and data L1 caches, and up to 65% for a unified L2 cache. We also showed that our Way Guard outperforms the recently proposed Way Halting scheme in saving energy.

8. REFERENCES

[1] B. Batson and T. N. Vijaykumar. Reactive-Associative Caches. *Proc. of Int'l Conf. on Parallel Architectures and Compilation Techniques*, 2001.

[2] A. Border and M. Mitzenmacher. Network application of bloom filters: A Survey. In *40th Annual Allerton Conference on Communication, Control, and Computing*, 2002.

[3] B. Calder, D. Grunwald, and J. Emer. Predictive sequential associative cache. *Proc. of the Int'l Symp. on High-Performance Computer Architecture*, 1996.

[4] Y.-J. Chang, S.-J. Ruan, and F. Lai. Sentry tag: An efficient filter scheme for low power cache. In *Proc. the 7th Asia-Pacific Computer Systems Architectures Conference*, 2002.

[5] S. Cohen and Y. Matias. Spectral bloom filters. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, 2003.

[6] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep packet inspection using parallel bloom filters. In *IEEE Hot Interconnects 12*, 2003.

[7] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[8] M. Ghosh, E. Özer, S. Biles, and H.-H. S. Lee. Efficient System-on-Chip Energy Management with a Segmented Bloom Filter. In *Proc. the 19th Int'l Conf. on Architecture of Computing Systems*, 2006.

[9] K. Inoue, T. Ishihara, and K. Murakami. Way-predicting set-associative cache for high performance and low energy consumption. In *Proc. 1999 International Symposium on Low Power Electronics and Design*, 1999.

[10] K. Lawton. Welcome to the Bochs x86 PC Emulation Software Home Page. http://www.bochs.com.

[11] G. Keramidas, P. Xekalakis, and S. Kaxiras. Applying Decay to Reduce Dynamic Power in Set-Associative Caches. In *2007 International Conference on High Performance Embedded Architectures and Compilers*, January 2007.

[12] R. Kessler, R. Jooss, A. Lebeck, and M. Hill. Inexpensive Implementations Of Set-Associativity. *The 16th Annual Int'l Symp. on Computer Architecture*, 1989.

[13] A. Kumar, J. Xu, J. Wang, O. Spatschek, and L. Li. Space-code bloom filter for efficient per-flow traffic measurement. In *Proc. IEEE INFOCOM*, March 2004.

[14] A. Ma, M. Zhang, and K. Asanovic. Way memoization to reduce fetch energy in instruction caches. *Workshop on Complexity Effective Design*, 2001.

[15] N. Mehta, B. Singer, R. I. Bahar, M. Leuchtenburg, and R. Weiss. Fetch halting on critical load misses. In *Proc the 22nd Int'l Conf. on Computer Design*, 2004.

[16] G. Memik, G. Reinman, and W. H. Mangione-Smith. Just say no: Benefits of early cache miss determination. In *Proceedings of the Ninth International Symposium on High Performance Computer Architecture*, 2003.

[17] A. Moshovos, G. Memik, B. Falsafi, and A. Choudhary. Jetty: Snoop filtering for reduced power in smp servers. In *Proceedings of International Symposium on High Performance Computer Architecture (HPCA-7)*, January 2001.

[18] J.-K. Peir, S.-C. Lai, S.-L. Lu, J. Stark, and K. Lai. Bloom filtering cache misses for accurate data speculation and prefetching. In *Proceedings of the 16th International Conference of Supercomputing*, pages 189–198, 2002.

[19] A. Roth. Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization. In *Proc. of the 32th Int'l Symp. on Computer Architecture*, 2005.

[20] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler. Scalable hardware memory disambiguation for high ilp processors. In *Proceedings of the 36th International Symposium for Microarchitecture*, 2003.

[21] D. H. Woo, M. Ghosh, E. Özer, S. Biles, and H.-H. S. Lee. Reducing energy of virtual cache synonym lookup using bloom filters. In *Proc. the 2006 Int'l Conf. on Compilers, Architecture and Synthesis for Embedded Systems*, 2006.

[22] C. Zhang, F. Vahid, J. Yang, and W. Najjar. A way-halting cache for low-energy high-performance systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2(1):34–54, 2005.

[23] M. Zhang and K. Asanovic. Highly-associative caches for low-power processors. In *Kool Chips Workshop in conj. with MICRO-33*, 2000.