

Reducing Energy of Virtual Cache Synonym Lookup using Bloom Filters

Dong Hyuk Woo Mrinmoy Ghosh Emre Özer† Stuart Biles† Hsien-Hsin S. Lee

School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332
{dhwoo, mrinmoy, leehs}@ece.gatech.edu

†ARM Ltd.
Cambridge, UK
{emre.ozet, stuart.biles}@arm.com

ABSTRACT

Virtual caches are employed as L1 caches of both high performance and embedded processors to meet their short latency requirements. However, they also introduce the synonym problem where the same physical cache line can be present at multiple locations in the cache due to their distinct virtual addresses, leading to potential data consistency issues. To guarantee correctness, common hardware solutions either perform serial lookups for all possible synonym locations in the L1 consuming additional energy or employ a reverse map in the L2 cache that incurs a large area overhead. Such preventive mechanisms are nevertheless indispensable even though synonyms may not always be present during the execution.

In this paper, we study the synonym issue using Windows applications workload and propose a technique based on Bloom filters to reduce synonym lookup energy. By tracking the address stream using Bloom filters, we can confidently exclude the addresses that were never observed to eliminate unnecessary synonym lookups, thereby saving energy in the L1 cache. Bloom filters have a very small area overhead making our implementation a feasible and attractive solution for synonym detection. Our results show that synonyms in these applications actually constitutes less than 0.1% of the total cache misses. By applying our technique, the dynamic energy consumed in L1 data cache can be reduced up to 32.5%. When taking leakage energy into account, the savings is up to 27.6%.

Categories and Subject Descriptors

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

General Terms

Design, Experimentation

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.

CASES'06, October 23–25, 2006, Seoul, Korea.

Copyright 2006 ACM 1-59593-543-6/06/0010 ...\$5.00.

Keywords

Cache, Synonym, Bloom filter, Low power

1. INTRODUCTION

Virtual caches offer advantages over physical caches because they do not need an address translation path from virtual addresses generated by the processor. This can potentially reduce the critical path to the cache by eliminating the address translation stage through the Translation Lookaside Buffer (TLB), and therefore the cache access time can be reduced significantly.

However, the major problem with the virtual caches is the *synonym* or *aliasing* problem [26] where multiple virtual addresses can map to the same physical address. The synonym problem occurs when the cache index bits use some of bits from the virtual page number without TLB translation. In such cases, the cache block may reside in multiple cache locations, and all the tags in these locations must be looked up. So, the tag lookup must be done in multiple cache lines for direct-mapped caches and in multiple cache sets for set-associative caches.

Synonyms can be avoided by either software, hardware or a combination of both. Software techniques involve OS intervention that restricts address mapping. For instance, page coloring [14] that aligns pages so that their virtual and physical addresses point to the same cache set. However, page coloring can have an effect on system performance such as page fault rate and restricts memory allocation. Another software solution to the synonym problem is that the OS flushes the caches on context switches to prevent synonyms, at a cost of degrading the overall system performance.

Hardware solutions, on the other hand, can detect synonyms without restricting the system software. Some of these solutions use duplicated tag arrays or reverse maps [11, 15, 26, 29], however, these techniques increase the die area and the power budget. Instead, designers prefer a simpler approach that only a single copy of a cache line is kept in the cache at any time [7, 27]. Synonyms are eliminated by the cache controller on a cache miss by searching for a possible synonym in every cache line belonging to the synonymous sets. In this paper, we assume that this simpler approach is used to detect synonyms.

Performing multiple tag lookups for finding a synonym consumes considerable amount of cache dynamic energy. The increasing dynamic energy consumed by tag lookups

256 sets that are indexed by 8 bits, 6 of which come from the page offset. The remaining 2 bits are taken from the virtual page number without being translated. Because of these 2 virtual bits, if different, the requested cache line can be in 4 different sets or 8 different lines in the 2-way cache.

When a cache request is made, the cache is accessed with the given virtual index and both ways are looked up for a hit. If there is a hit, then this means that there is no synonym in the cache as the cache keeps a single copy. If there is a miss, this may or may not be a true miss because the requested cache line might be in the other 3 synonym cache sets. The miss request is sent to the L2 cache and in the mean time, the tag lookup in the L1 continues serially until a synonym is found. After 3 serial lookups, if there is a hit, the synonym is found in the cache, and the lookup in the L1 is stopped and the miss request to the L2 cache is also aborted. Then, the synonym is moved to the location that generated the miss.

If a synonym is not found after 3 serial lookups, this is a true miss and the cache does not have a synonym with the given address. As such, 6 additional tag lookups were performed to determine if there is a real miss. This certainly increases the cache dynamic energy consumption.

Given the fact that synonyms are infrequent, performing the additional search for a synonym is generally a waste of dynamic energy. We can eliminate the extra synonym lookups if we know for certain that there is no synonym in the cache. For this purpose, we propose to use a Bloom filter associated with the cache that can safely indicate that there is no synonym in the cache. The next section describes the philosophy behind Bloom filters.

2.2 Bloom Filters

Bloom filters were proposed by Burton Bloom in his seminal paper [2]. The original purpose was to build memory efficient database applications. The structure of a generic Bloom filter is shown in Figure 3(a). The first design is the direct implementation of the Bloom filter theory where the filter vector is constantly inserted without deletion. A given address in N bits is hashed into k hash values using k different hash functions. The output of each hash function is an m -bit index value that addresses the Bloom filter bit vector of 2^m . Here, m is much smaller than N . Each element of the Bloom filter bit vector contains only one bit that can be set. Initially, the Bloom filter bit vector is set to zero. Whenever an N -bit address is observed, it is hashed to the bit vector and the bit values hashed by each m -bit index are set to one.

When a query is to be made whether a given N -bit address has been observed before, the 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 values. If at least one of the bit values is 0, this means that this address has definitely not been observed before. This is also called a *true miss*. If all of the bit values are 1, then the address may have been observed but the filter cannot guarantee that it has been definitely observed. Nevertheless, if the address has actually not been observed but the filter indicates 1, then this case is called a *false hit*.

As the number of hash functions increases, the Bloom filter bit vector is polluted much faster. On the other hand, the probability of finding a zero during a query increases if more hash functions are used.

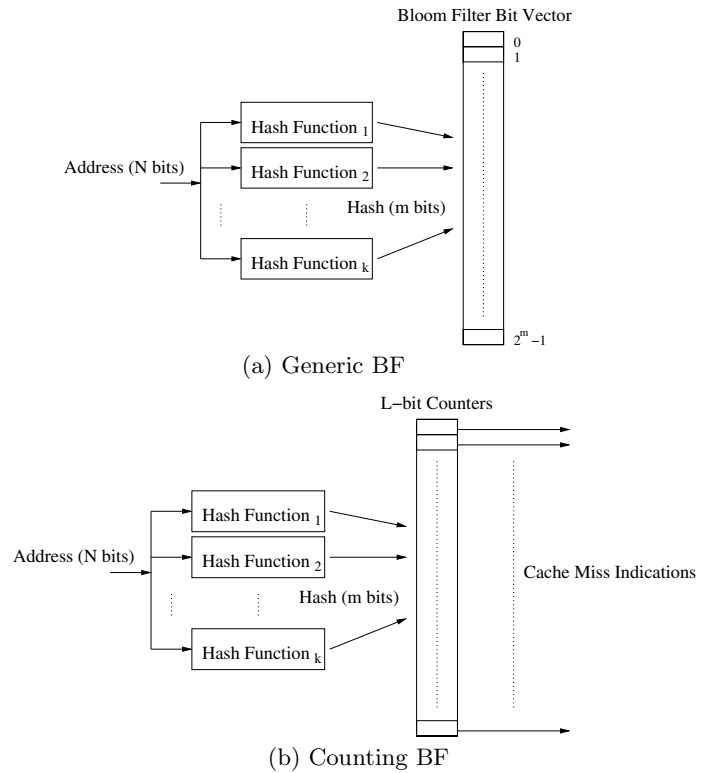


Figure 3: Bloom Filter

The major drawback of the original Bloom filter is that the filter can be polluted rapidly and filled up with all 1s as it does not have deletion capability. Thus, the counting Bloom filter was proposed to allow deleting entries from the filter as shown in Figure 3(b). The counting Bloom filter reduces the number of false hits by introducing counters instead of a bit vector. It had been originally proposed for web cache sharing in [9]. In the counting Bloom filter, when a new address is observed for addition to the Bloom filter, each m -bit hash index addresses to a specific counter in an L -bit counter array.¹ Then, the counter is incremented by one. Similarly, when a new address is observed for deletion from the Bloom filter, each m -bit hash index addresses to a counter, and then the counter is decremented by one. If more than one hash index addresses to the same location for a given address, the counter is incremented or decremented only once. If the counter is zero, it is a true miss. Otherwise, the outcome is not precise.

3. MISS TRACKING WITH BLOOM FILTERS

Our Bloom filter design contains three main structures: a counter array having 2^m counters of L bits, a bit vector of 2^m entries and one hash function. We use only a single hash function because our experiments show that the filtering rate of a Bloom filter with a single hash function is not awfully worse than the one with many hash functions. The

¹ L must be wide enough to prevent saturations.

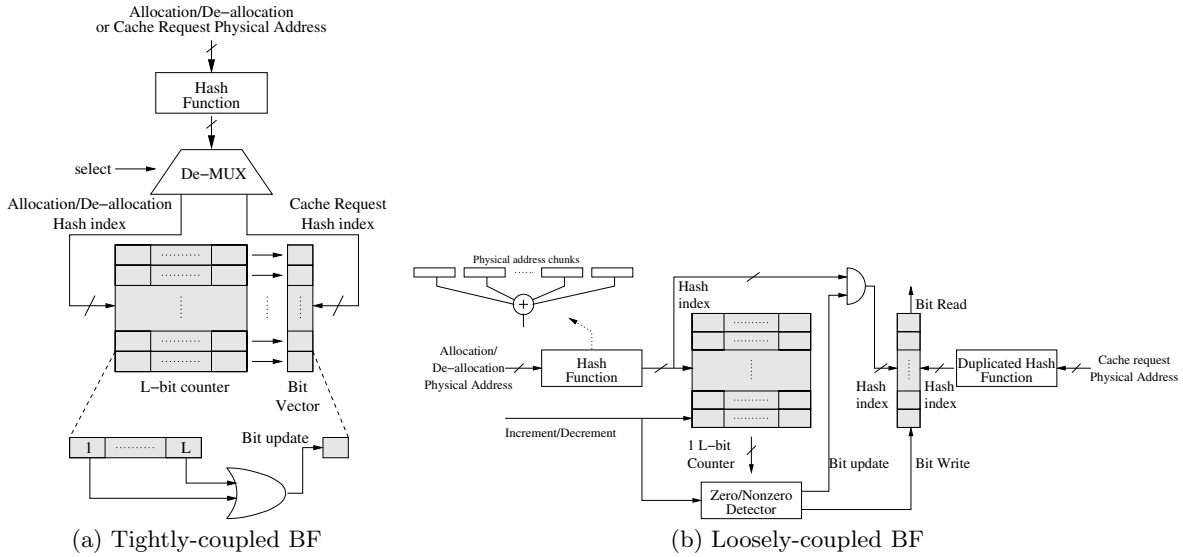


Figure 4: Cache miss tracking with a Bloom Filter

implemented hash function chops the physical address into several chunks of hash index long and bitwise XOR them to obtain a single hash index as shown in Figure 4(b). The value of L depends on the hash function that distributes indexes across the Bloom filter. In the worst case, if all cache lines map to the same counter, the bit-width of the counter must be at most $\log_2(\#of\ cachelines)$ to prevent overflows. In reality, the required number of bits per counter is much smaller than the worst-case.

The hash function reads a *physical address* to generate m -bit hash index. The physical address can be an allocation and de-allocation address from the lower level memory or a cache request address from the CPU. When the CPU requests a line from the cache, it also sends the request address to the bit vector. The hash function hashes the address, generates an index and reads the bit value from the vector. If the value is 0, this is a safe indication that this address has *never* been observed before. If it is 1, it is an indefinite response in the sense that the address may or may not have been observed before.

The counter array is updated with cache line allocation and de-allocations. Whenever a new cache line is allocated, the address of the allocated line is hashed into the counter array and the associated counter is incremented. Similarly, when a cache line is evicted from the cache, its counter associated with the hashed physical address of the line is decremented. The order and timing of these two operations depend on the implementation of the cache and its controller. The counter array is responsible for keeping the bit vector up-to-date.

3.1 Tightly-coupled Counting Bloom Filter

Figure 4(a) depicts a cache miss tracking mechanism using a tightly-coupled counting Bloom filter. The tightly-coupled Bloom filter design allows instantaneous updates to the bit vector from the counter array. The de-multiplexer forwards the hash index generated by the hash function to

the counter array or to the bit vector depending on the type of the physical address. In the case of a cache line allocation or de-allocation, the m -bit hash index is sent to the L -bit counter array to index to a specific counter, and then the counter is incremented or decremented.

The bit vector is updated at every cycle using a logical L input OR gates per counter. For each counter in the array, the L -bits are wired to an OR gate to generate the bit update to the selected bit vector location.

3.2 Loosely-coupled Counting Bloom Filter

The main advantage of the tightly-coupled Bloom filter is that the counter array can update the bit vector instantaneously. However, the updates to the counters are more often than updates to the bit vector. In one of these infrequent updates, only one location in the bit vector is written. Also for querying the Bloom filter only the bit vector is needed. Thus having to query the large tightly coupled structure is overly expensive.

Thus, we propose an innovative loosely-coupled counting Bloom filter as shown in Figure 4(b) where the counter array is decoupled from the bit vector and the hash function is duplicated on the bit vector side. The allocation/de-allocation addresses are sent to the counter array using one hash function while the cache request address from the CPU is sent to the bit vector using the copy of the same hash function. By decoupling these two structures, the counter array can be run with lower frequency, the size of which is much bigger than the bit vector. This will help to save dynamic energy.

The update from the counter array to the bit vector is done only for a single bit location. First, the L -bit counter value is read from the counter array before an increment or decrement operation is done. Then, the counter value is checked for a zero boundary condition by the zero/nonzero detector whether it will become non-zero from zero or zero from non-zero, which will be inferred by the increment / decrement line.

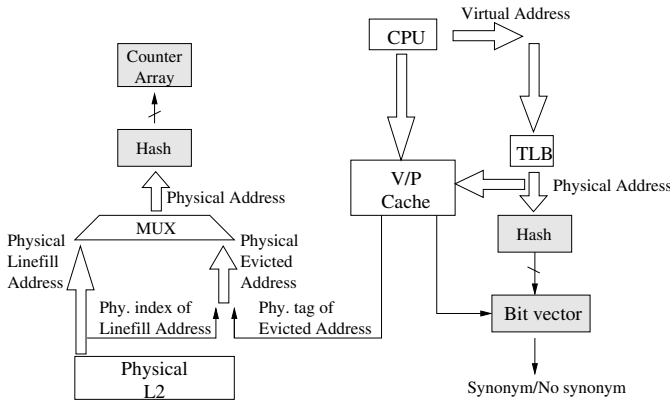


Figure 5: Early synonym detection in a V/P cache

If a zero boundary condition is detected, the bit update line is activated, which enables the hash index to be forwarded to the bit vector. Finally, the bit write line is made 1 to set the bit vector location if the counter has become non-zero. Otherwise, the bit write line is made 0 to reset the bit vector location. If there is no zero boundary condition, then the bit update line is not activated, which disables the hash index forwarding to the bit vector.

With the loosely-coupled counting Bloom filter design, the counter array and bit vector can be located in separate physical locations.

4. EARLY SYNONYM DETECTION WITH A DECOUPLED BLOOM FILTER

If finding synonyms is a rare event per cache miss, then the Bloom filter can be quite successful for fast and low-power detection of synonyms in virtual caches. It is fast because it can tell us whether there is no synonym by checking a single bit location in the vector. In contrast, the baseline serial lookup cache detection may also be detrimental to performance in some cases.² It is low-power because $(w \cdot S)$ tag lookup comparisons can be eliminated where w is the cache associativity and S is the number of synonym sets. The frequency of synonyms depends on the number of shared pages between the kernel and application codes. When also considering the fact that cache misses are less frequent than hits, the frequency of finding synonyms per cache miss is likely to be low, so the virtual cache system certainly benefits from using a Bloom filter to filter out several tag lookups.

4.1 Virtually-indexed Physically-tagged Caches

Figure 5 shows early synonym detection mechanism with a loosely-coupled Bloom filter for a virtually-indexed physically-tagged cache. The Bloom filter keeps track of physical addresses rather than virtual ones so that it can provide a quick response regarding synonym existence with a given physical address. The counter array is updated with the

²If a cache has 3 synonym bits it would have 8 possible locations to search for a synonym. If each search takes 2 cycles and the L2 cache access time is 10 cycles, searching for a synonym serially is worse than going to the L2 cache.

physical addresses of the linefill from the physical L2 cache and the evicted line. The physical address of the evicted line is formed from the tag that contains the usual tag bits and the translated synonym bits, appended with the lower part of the index of the set.

When there is a miss in the V/P cache for a lookup, the physical address is sent to the L2 cache to service the miss. At the same time, the physical address is hashed into the bit vector to check for any synonym. If the bit location is zero, there is no synonym in the V/P cache. In that case, no synonym lookup is necessary in the cache. If the bit location is 1, this implies that there may be a synonym in the cache, and a serial synonym lookup process is launched to all synonym sets for synonym detection.

4.2 Virtually-indexed Virtually-tagged Caches

For virtually-indexed virtually-tagged caches, the Bloom filter counters are updated by the physical address of the linefill during a linefill. For eviction, however, the evicted address must be translated into the physical address by the TLB. The dirty evicted lines are already translated by the TLB before sending it to the physical memory. The only extra TLB translations are needed for the clean evicted lines. When a cache miss occurs, the Bloom filter vector is accessed by the physical address acquired from the TLB. Similar to the virtually-indexed physically-tagged caches, a true miss in the Bloom filter vector (value returned is zero) means that there is no synonym in the cache associated with this physical address, and a possible hit (value returned is one) means that all synonym sets must be looked up for a synonym in the cache. This involves in translating all virtual tags in all the synonym sets into physical tags and then compare them to the physical tag of the outstanding address.

5. EXPERIMENTAL RESULTS

5.1 Simulation Environment

We use Bochs [13] to perform full-system simulation. The simulation involves running one or more common Windows applications on the Windows NT platform. To collect cache statistics, we integrate SimpleScalar’s cache simulator in Bochs. We also enhance the cache simulator with our Bloom filter model. This innovative simulation technique allows us to gather cache statistics of various applications running on top of a full operating system like Windows or Linux. Previous techniques [28] using Bochs, collected instruction execution traces from Bochs, that were then fed into a micro-architectural simulator like SimpleScalar. By integrating the cache simulator into Bochs we remove the time-consuming and cumbersome process of trace collection from the simulation process. This technique is much faster than trace collection and we may simulate a lot more instructions than trace collection techniques, as trace collection inherently limits the size of the trace that may be collected.

We assume that the size of each page is 4 KB and use 16 different configurations for L1 data cache as shown in Table 1. These 16 cache configurations are the only configurations possible for cache sizes less than 64KB with a page size of 4KB to have the synonym problem.³ All cache configurations are virtually indexed physically tagged. The L1

³We exclude four configurations with 128 sets that Artisan 90nm SRAM library does not support.

Table 1: L1 data cache configuration

# of sets	# of ways	Line size (B)	Cache size (KB)	Bit vector (b)	Counter array (b)	s-bits	S^{**}
256	1	32	8	1024	3072	1	2
256	2	32	16	2048	6144	1	2
256	4	32	32	4096	12288	1	2
256	8	32	64	8192	24576	1	2
512	1	32	16	2048	6144	2	4
512	2	32	32	4096	12288	2	4
512	4	32	64	8192	24576	2	4
1024	1	32	32	4096	12288	3	8
1024	2	32	64	8192	24576	3	8
2048	1	32	64	8192	24576	4	16
256	1	64	16	1024	3072	2	4
256	2	64	32	2048	6144	2	4
256	4	64	64	4096	12288	2	4
512	1	64	32	2048	6144	3	8
512	2	64	64	4096	12288	3	8
1024	1	64	64	4096	12288	4	16

* Size of one counter is 3 bits

** S is the number of synonym sets

Table 2: Simulation workload

Workload	Instruction count	Description
<i>Winamp</i>	1.9 B	Play a .mp3 file (7MB)
<i>Internet Explorer</i>	2.0 B	Visit several famous websites every five seconds
<i>Winzip</i>	2.1 B	Decompress a .zip file (4MB)
<i>Word</i>	7.2 B	Open a .doc file (4MB) and edit it
<i>Excel</i>	1.0 B	Open a .xls file (4MB) and edit it
<i>PowerPoint</i>	12.4 B	Open a .ppt file (8MB) and edit it
<i>VC++</i>	29.2 B	Compile Bochs 2.2.5 and execute it
Two app.	38.6 B	<i>VC++</i> and <i>Winamp</i>
Three app.	40.5 B	<i>VC++</i> , <i>Winamp</i> and <i>Internet Explorer</i>

instruction cache has also been modeled and it is taken to be always same as the L1 data cache configuration. To estimate power consumption for the caches and the Bloom filter, we use Artisan 90nm SRAM library. The Artisan SRAM generator is capable of generating synthesizable Verilog code for SRAMs in 90nm technology. A datasheet is also generated that gives an estimate of the read and write power of the generated SRAM. The datasheet also provides a standby current from which we can estimate the leakage power of the SRAM.

The energy reduction reported are only for the data cache.⁴ Our simulation system runs Windows NT and we use seven Windows applications to evaluate our approach as shown in Table 2. Running the Windows applications involved doing simple tasks like opening a *PowerPoint* file and modifying it. Since it would be difficult to exactly reproduce actions that involve human interaction like clicking the mouse, we decided to simulate all 16 cache configurations simultaneously. We instantiated 16 cache simulators and fed them with the instruction stream produced by Bochs. The total number of instructions executed ranges from one billion to 41 billions. Throughout all simulations that we performed, we evaluate the advantage of our Bloom filter enhanced

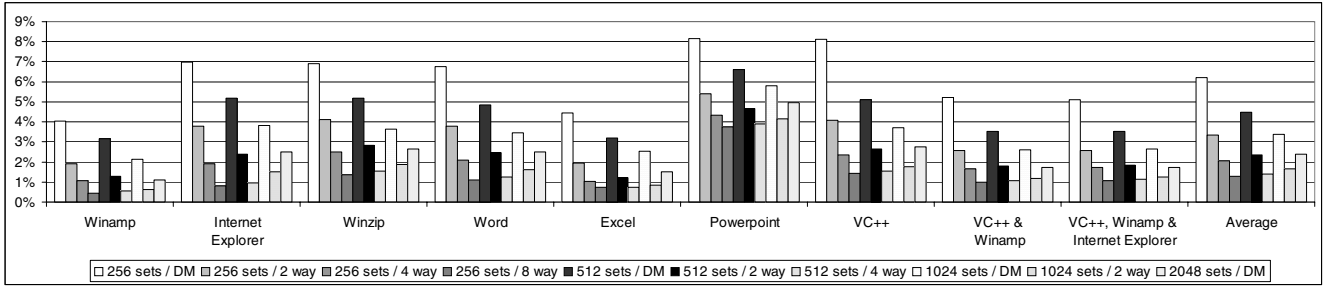
⁴Synonyms in I-Cache do not affect correctness as they are read only. Also hit rates of I-Cache is extremely high. So we assume no hardware technique for detecting synonyms for the I-Cache.

cache structure compared with the conventional cache structure.

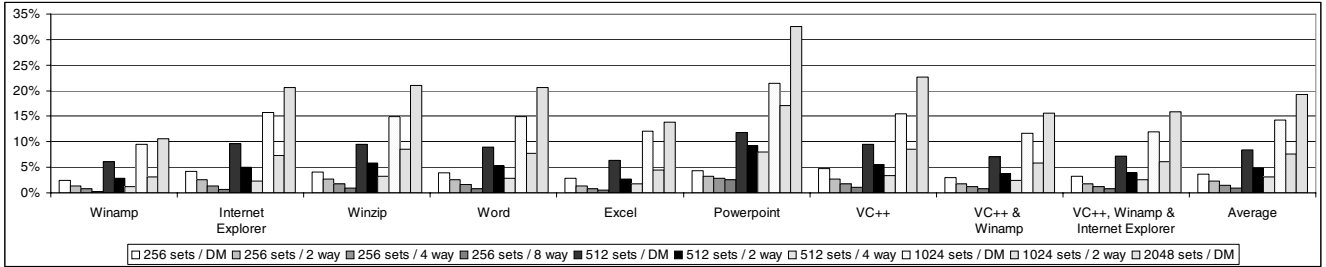
We accurately model the delay and energy overhead for the Bloom filter. To estimate dynamic and static power consumption we synthesize the Bloom filter structures using the same 90nm Artisan SRAM generator used for synthesizing the caches. Since we have implemented the loosely coupled design, the bit vector and the counter array are synthesized separately. We add the bit vector access energy every time the Bloom filter is queried for the presence or absence of an address. This is done on every cache miss to check for synonyms. The counter array access energy is added every time the structure is updated. This happens when a cache line is evicted from the cache and also when a line-fill occurs. Since our bit vector is a relatively small structure than the cache, we assume the access latency to be one cycle. However this access latency affects performance only on cache miss events in the presence of synonyms. Since the occurrence of synonyms is very rare it has negligible effect on the performance of the system.

5.2 Results

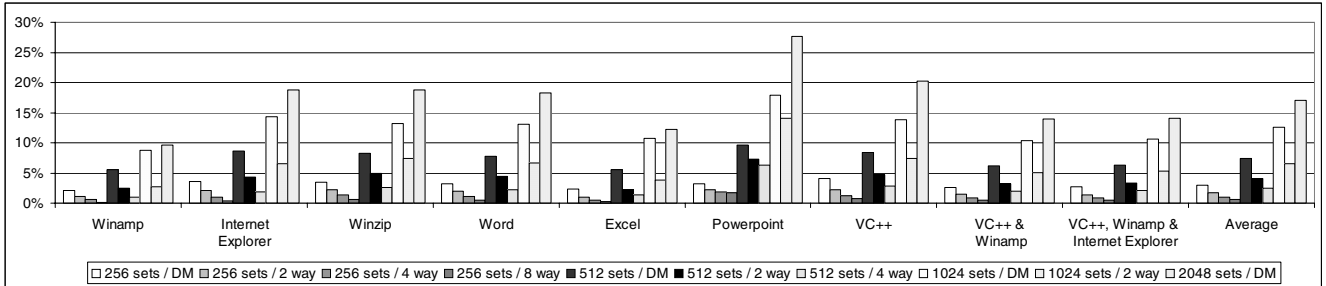
First, we analyze the number of synonyms in common Windows applications. All our nine simulations show that synonyms are rarely present. The average synonym ratio is 0.08%, meaning that only eight out of ten thousands cache misses are identified as synonyms. Note that, even though



(a) Miss rate



(b) Dynamic energy reduction



(c) Total energy reduction

Figure 6: Overall Simulation Results for 32B-Line Caches

the occurrence of synonyms is rare, it is nonetheless required for the baseline processor to examine all synonym sets on every cache miss — leading to large dynamic energy consumption just for synonym detection. This observation justifies our motivation for finding a lighter weight alternative in the hardware to alleviate the synonym problem.

One reason of the synonym ratio being extremely low is that the number of operations between successive context switches is very large. Since the L1 data cache is very small, the cache would only contain data pertaining to the presently running process. This reduces the probability of having shared data from another process, making the number of synonyms negligible. Synonyms would only be detected at the initial phase of one context switching period, after which the cache would be warmed up and filled with data from the current process.

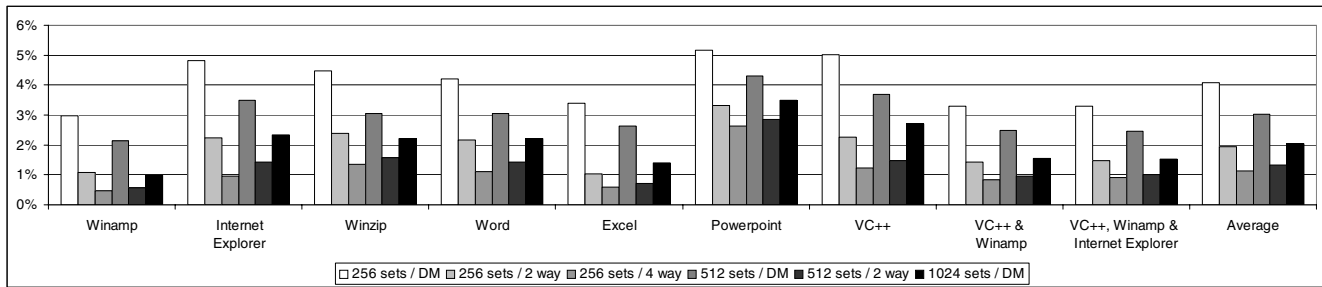
As shown in Figure 6(b) for 32B cache line cases, the amount of dynamic energy reduction ranges from 0.3% to 32.5% depending on the cache configuration and the characteristics of application. These savings take into account the energy overhead of the bit vector and the counter array.

The overhead on the average was found to be 0.22% over all the benchmark programs. Similar trends are observed in caches with 64B lines shown in Figure 7(b).

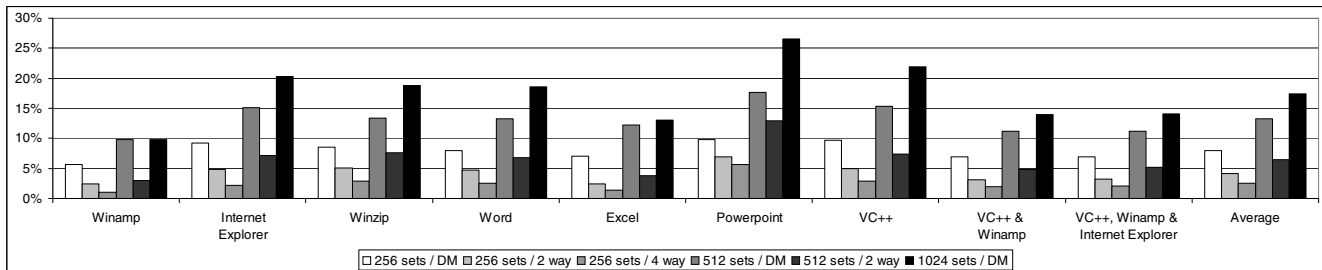
To further understand the source of energy savings, we use *PowerPoint* running on 32-byte line caches for our analysis. Figure 8(a) plots the normalized dynamic energy consumed by hits and misses in the baseline cases, and by true misses, false hits in the Bloom filter enhanced caches. We may easily see that miss handling contributes to a significant amount of L1 data cache dynamic energy by up to 47.0% (for the configuration of 2048 sets, Direct-Mapped). The figure also shows that using Bloom filters, dynamic energy consumption by such miss handling can be significantly reduced, resulting in an overall 32.9% energy reduction for this configuration.

Figure 6(c) shows that the amount of total energy reduction (including both dynamic and leakage energy) of L1 data cache ranges between 0.1% and 27.6%.⁵ This savings take into account the leakage energy consumption of the added

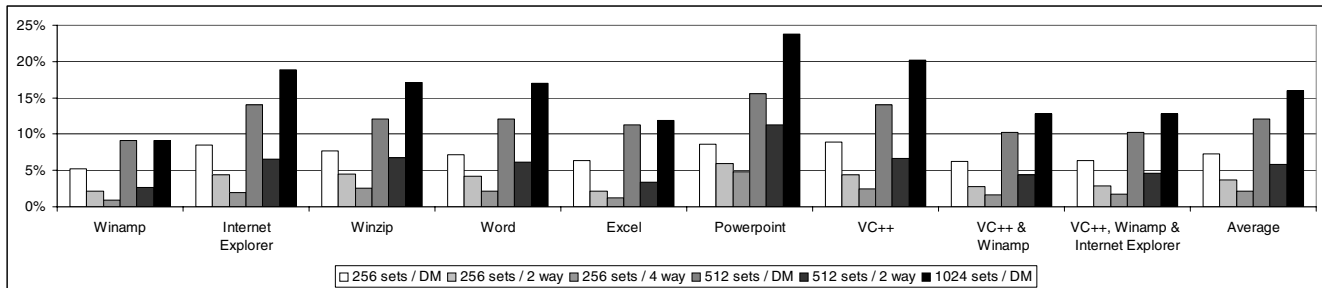
⁵To facilitate the estimation of the leakage energy, we assume a 2GHz in-order processor with blocking cache. The latencies of L1, L2 and DRAM are 2, 10 and 250 cycles.



(a) Miss rate



(b) Dynamic energy reduction



(c) Total energy reduction

Figure 7: Overall Simulation Results for 64B-Line Caches

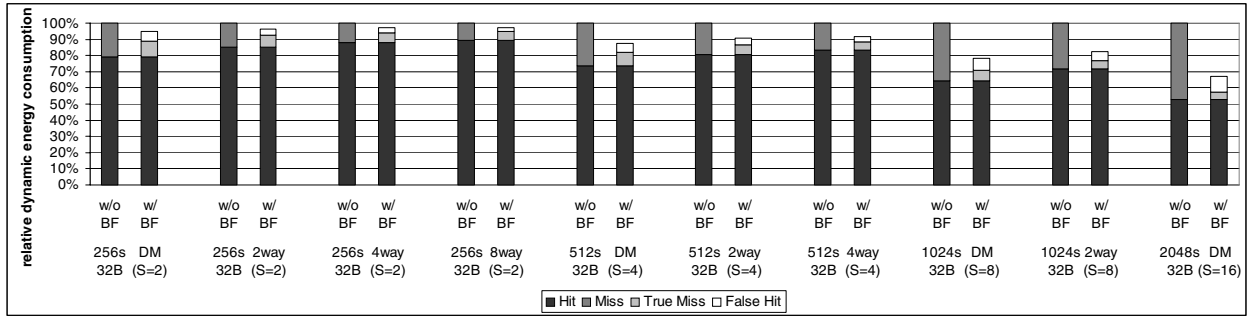
hardware structures, namely the Bloom filter and the Counters. The average leakage energy overhead of the additional hardware was found to be 0.31% over all the benchmarks. As shown in Figure 6(a), the energy saving of a workload is highly dependent on its miss rate. All our workloads have miss rates lower than 9%. We expect higher L1 miss rates to be present in commercial workloads, where our technique will render larger benefit.

Figure 8(b) shows the relative percentage of L1 cache accesses using our technique compared to the baseline cache. In this figure for every cache miss we count S accesses to the cache, where S is the number of synonym sets. We can see that using our technique we may reduce cache accesses by

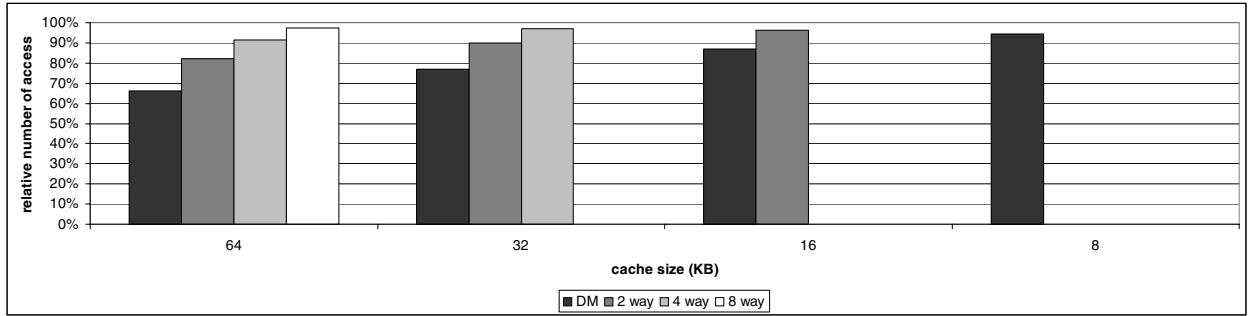
We assume the average micro-ops to x86 instruction ratio is 1.4 as shown in [25]. Note that our approach is very conservative because longer execution time of our simulation model will lead to exaggerated leakage energy consumption although our solution is aimed for reducing dynamic energy consumption. Thus, the actual total energy reduction using our scheme in state-of-the-art processors is expected to be higher.

up to 33.9%. We know that L1 cache performance is critical to overall system performance. The contention for the L1 cache would be more of a critical factor in superscalar and SMT systems where multiple cache accesses occur in a single cycle. Our simple hardware technique would be extremely useful in such systems as it would reduce a large number of accesses and contention for the L1 cache.

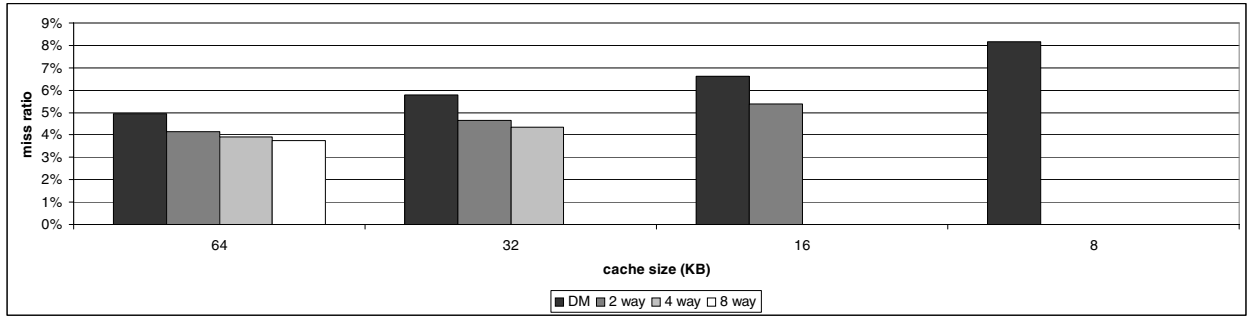
Figure 8(d) shows the effect of cache configuration on the effectiveness of our approach. From the figures we see that energy reduction decreases when associativity increases. One reason is because increasing associativity drastically reduces the number of misses as shown in Figure 8(c) and thus the opportunity of our technique to save energy. In addition, as the associativity increases, the number of synonym sets drops linearly in cache configurations having the same cache and line sizes. The design trend of modern L1 caches, however, are geared toward lowering associativity to meet cycle time constraints. Another observation that can be made from these figures is that energy reduction decreases as cache size is reduced. This is because even though miss rates decrease with larger caches, the number of synonym sets also



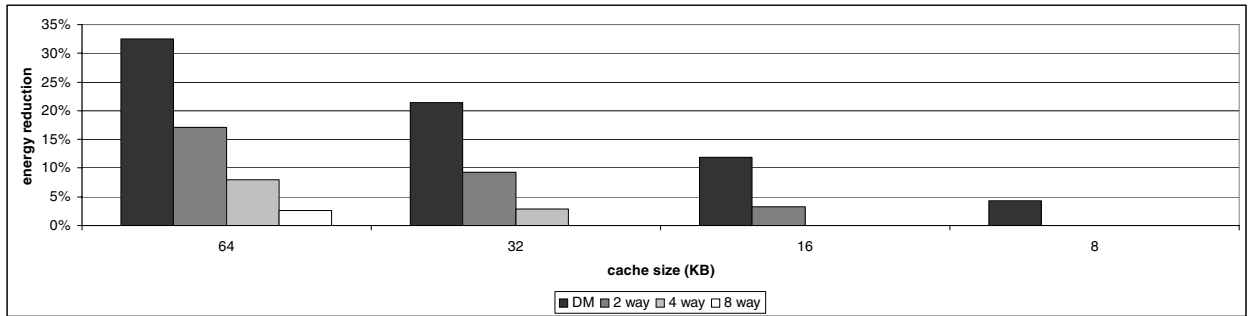
(a) Relative dynamic energy consumption



(b) Relative number of L1 data cache accesses



(c) Miss rate



(d) Dynamic energy reduction

Figure 8: Simulation Results of *PowerPoint* for 32B-Line Caches

increase. This gives a significant energy saving opportunity for our technique over the baseline.

6. RELATED WORK

Bloom filters have found numerous uses in networking and database applications such as [3, 5, 6, 8, 16, 22]. They are also used as microarchitectural blocks for tracking load/store addresses for resolving load-store conflicts, and also for performing load-store queue optimizations [1, 23, 24]. Moshovos et al. [19] uses a Bloom filter to filter out cache snoops in SMP systems. Peir et al. [21] detects load misses early in the pipeline for speculative execution. Mehta et al. [17] also uses it to detect L2 misses early so that the instruction fetch can be stalled early to save processor energy. Memik et al. [18] proposes Bloom filter-like early cache miss detection techniques for reducing dynamic cache energy and to improve the performance by bypassing them. More recently, Ghosh et al. [10] proposed to use counting Bloom filters to detect L2 misses a priori for energy reduction in L2 accesses.

None of the techniques above discuss the energy consumption problem caused by synonym lookup in virtual caches or propose any microarchitectural solutions.

7. CONCLUSIONS

Implementing virtual caches can accelerate L1 cache accesses while introducing the synonym problem. Even though a synonym hit is an infrequent event, to guarantee correctness, the processor still needs to perform additional lookups for all possible synonym locations upon each L1 miss, thereby consuming more energy. In this paper, we propose an early synonym detection mechanism using Bloom filters. It is shown to be fast, effective, and consume lower power. By tracking and checking the address signature in the filters, we are able to exclude the unnecessary lookups for addresses that were never accessed.

Using x86 applications based on Bochs, our simulation results show that our decoupled Bloom filter mechanism can effectively reduce the total cache accesses by up to 33.9%. The dynamic energy consumption and the overall energy consumption (including leakage energy) in the L1 can be reduced by up to 32.5% and 27.6%, respectively.

8. ACKNOWLEDGMENT

This research was supported in part by National Science Foundation Grants CCF-0326396 and CNS-0325536.

9. REFERENCES

- [1] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proceedings of IEEE/ACM 36th International Symposium on Microarchitecture*, 2003.
- [2] B. Bloom. Space/Time Tradeoffs in Hash Coding with Allowable Errors. *Communications of the ACM*, 1970.
- [3] A. Border and M. Mitzenmacher. Network Application of Bloom Filters: A Survey. In *Proceedings of the 40th Annual Allerton Conference on Communication, Control, and Computing*, 2002.
- [4] M. Cekleov and M. Dubois. Virtual-Address Caches Part 1: Problems and Solutions in Uniprocessors. *IEEE Micro*, 1997.
- [5] F. Chang, W. Feng, and K. Li. Approximate Caches for Packet Classification. In *INFOCOM'04*, 2004.
- [6] S. Cohen and Y. Matias. Spectral Bloom Filters. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, 2003.
- [7] Compaq Computer. *Alpha 21264 Microprocessor Hardware Reference Manual*.
- [8] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep Packet Inspection using Parallel Bloom Filters. In *Hot Interconnects 12*, 2003.
- [9] L. Fan, P. Cao, J. Almerda, and A. Broder. Summary cache: A scalable wide-area Web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 2000.
- [10] Mrinmoy Ghosh, Emre Ozer, Stuart Biles, and Hsien-Hsin S. Lee. Efficient System-on-Chip Energy Measurement with a Segmented Bloom Filter. In *Proceedings of the 19th International Conference on Architecture of Computing Systems*, pages 283–297, 2006.
- [11] James R. Goodman. Coherency for Multiprocessor Virtual Address Caches. In *Proceedings of the 2nd Symposium on Architectural Support for Programming Languages and Operating Systems*, 1987.
- [12] K. Inoue, T. Ishihara, and K. Murakami. Way-Predicting Set-Associative Cache for High Performance and Low Energy Consumption. In *ISLPED-99*, 1999.
- [13] K. Lawton. Welcome to the Bochs x86 PC Emulation Software Home Page. <http://www.bochs.com>.
- [14] R. E. Kessler and M. D. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems*, 1992.
- [15] J. Kim, S. L. Min, S. Jeon, B. Ahn, D.-K. Jeong, and C. S. Kim. U-cache: A Cost-effective Solution to the Synonym Problem. In *Proceedings of the First Annual Symposium on High Performance Computer Architecture*, 1995.
- [16] A. Kumar, J. Xu, J. Wang, O. Spatschek, and L. Li. Space-Code Bloom Filter for Efficient Per-Flow Traffic Measurement. In *INFOCOM'04*, 2004.
- [17] N. Mehta, B. Singer, Iris Bahar, M. Leuchtenburg, and R. Weiss. Fetch Halting on Critical Load Misses. In *Proceedings of the 2004 IEEE International Conference on Computer Design*, 2004.
- [18] G. Memik, G. Reinman, and W. H. Mangione-Smith. Just Say No: Benefits of Early Cache Miss Determination. In *Proceedings of the Ninth Annual Symposium on High Performance Computer Architecture*, 2003.
- [19] A. Moshovos, G. Memik, B. Falsafi, and A. Choudhary. JETTY: Snoop filtering for reduced power in SMP servers. In *HPCA-7*, 2001.
- [20] J.-K. Peir, W. W. Hsu, and A. J. Smith. Implementation Issues in Modern Cache Memory. *IEEE Transactions on Computers*, 1999.
- [21] Jih-Kwon Peir, Shih-Chang Lai, Shih-Lien Lu, Jared Stark, and Konrad Lai. Bloom Filtering Cache Misses for Accurate Data Speculation and Prefetching. In *Proceedings of the 2002 International Conference on Supercomputing*, 2002.

- [22] S. Rhea and J. Kubiatowicz. Probabilistic Location and Routing. In *Proceedings of the IEEE INFOCOM'02*, 2002.
- [23] Amir Roth. Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, 2005.
- [24] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler. Scalable Hardware Memory Disambiguation for High ILP Processors. In *Proceedings of IEEE/ACM 36th International Symposium on Microarchitecture*, 2003.
- [25] B. Slechta, B. Fahs, D. Crowe, M. Fertig, G. Muthler, J. Quek, F. Spadini, S. Patel, and S. Lumetta. Dynamic optimization of micro-operations. In *Proceedings of the Ninth Annual Symposium on High Performance Computer Architecture*, 2003.
- [26] Alan J. Smith. Cache Memories. *ACM Computing Surveys*, September 1982.
- [27] S. T. Tucker. The IBM 3090 System: An Overview. *IBM Systems Journal*, 1986.
- [28] Stevan Vlaovic and Edward S. Davidson. TAXI: Trace Analysis for X86 Interpretation. In *Proceedings of the 2002 IEEE International Conference on Computer Design*, 2002.
- [29] Wen-Hann Wang, Jean-Loup Baer, and Henry M. Levy. Organization and Performance of a Two-Level Virtual-Real Cache Hierarchy. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 1989.