

Applying Decay to Reduce Dynamic Power in Set-Associative Caches

Georgios Keramidas¹, *Polychronis Xekalakis², and Stefanos Kaxiras¹

¹ Department of Electrical and Computer Engineering, University of Patras, Greece

² Department of Informatics, University of Edinburgh, United Kingdom
{keramidas,kaxiras}@ee.upatras.gr, p.xekalakis@ed.ac.uk

Abstract. In this paper, we propose a novel approach to reduce dynamic power in set-associative caches that leverages on a leakage-saving proposal, namely Cache Decay. We thus open the possibility to unify dynamic and leakage management in the same framework. The main intuition is that in a decaying cache, dead lines in a set need not be searched. Thus, rather than trying to predict which cache way holds a specific line, we predict, for each way, whether the line could be live in it. We access all the ways that possibly contain the live line and we call this way-selection. In contrast to way-prediction, way-selection cannot be wrong: the line is either in the selected ways or not in the cache. The important implication is that we have a fixed hit time — indispensable for both performance and ease-of-implementation reasons. In order to achieve high accuracy, in terms of total ways accessed, we use Decaying Bloom filters to track only the live lines in ways — dead lines are automatically purged. We offer efficient implementations of such autonomously Decaying Bloom filters, using novel quasi-static cells. Our prediction approach grants us high-accuracy in narrowing the choice of ways for hits as well as the ability to predict misses — a known weakness of way-prediction.

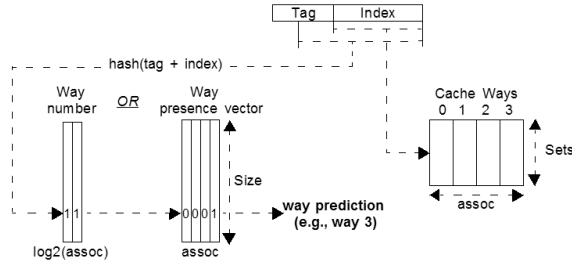
1 Introduction

Power consumption is a prime concern for the design of caches and numerous techniques have been proposed to reduce either leakage [9, 14, 19] or dynamic power [11, 12, 25, 26] but without a clear understanding of how to integrate them in a single implementation. We propose a new approach for reducing dynamic power in set-associative caches by exploiting a previous leakage-saving proposal. Specifically, we use Cache Decay [14] to identify dead cachelines, which we then exclude from the associative search.

Our proposal is different from way-prediction proposals that precede it. We do not try to predict a single way for an access but we base our approach on information about what is live and what is dead in the cache. In effect, instead of predicting a way number, we predict whether an access refers to a live line. We

* This work has been conducted while Polychronis Xekalakis was studying at the University of Patras, Greece.

Fig. 1. Way-Prediction implemented as an array of way-number predictions or as way-presence vectors with at most one bit set per vector.



make this prediction for each cache way and we access only the ways that can possibly hold the line in a *live state* — we call this *way-selection*. Way-selection cannot be wrong: the accessed line is either among the selected lines or it is a miss. As a result, we have a *fixed hit time* as opposed to way-prediction. This is a significant advantage in terms of implementation complexity (with respect to cache pipelining) but more importantly, in terms of performance [8]. Variable hit latency in L1 not only slows down the cache, but creates many difficulties in efficiently scheduling depended instructions in the core.

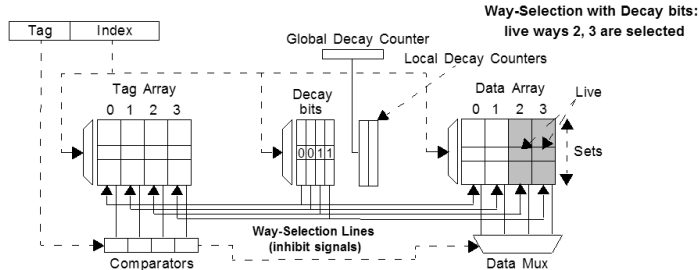
To achieve high-prediction accuracy, we use a special form of membership-testing hash tables called Bloom filters [3] to track liveness information for individual lines, per cache way. A normal Bloom filter (per way) would tell us whether a line possibly appeared before in a specific way, but without accounting for the possibility of the line being already dead. By decaying Bloom filter entries in concert with corresponding lines, a Decaying Bloom filter reflects the presence of the live lines only.

A significant benefit of decaying Bloom filter entries is the early prediction of cache misses. This is due to the Bloom filters reflecting the presence of live data in the cache and responding negatively to accesses that are going to miss. Early miss prediction is a boost to both performance and power savings since the cache can be bypassed completely on misses [8].

For simple and efficient implementations we propose and study various forms of Decaying Bloom filters, including Self-Decaying Bloom filters which decay their entries autonomously and require no decay feedback from the cache. To support power-efficient implementations of self-decaying Bloom filters we propose novel quasi-static cells designed to offer the necessary decay functionality for Bloom filters

Structure of the Paper: Section 2 motivates the need for a new approach and presents our proposal. Section 3 discusses details for efficient implementations. Section 4 presents our methodology and Section 5 our results. Section 6 reviews previous work and Section 7 concludes with a summary

Fig. 2. Decay and Way-Selection using Decay bits.



2 Way-Selection and Decaying Bloom Filters

2.1 What is wrong with Way-Prediction?

Way-prediction was initially proposed to address latency in set-associative caches [2, 4, 5]. These techniques also lead to power savings since they circumvent the associative search [10, 11, 13, 25, 26]. Way-prediction techniques aim to predict a single way where a reference might hit. Figure 1 shows the general view of way-prediction for a 4-way set-associative cache. For an n -way cache, the way-predictor can be viewed either as an array of $\log_2(n)$ -bit binary numbers or a n -bit presence vector.

A correct prediction results in a *fast* hit and yields power benefits roughly proportional to the associativity (a single way out of n is accessed). On the other hand, a way misprediction, results in no power savings, and a slow hit, not only because the rest of the ways need to be searched with a subsequent cache access but also because the instructions that depend on the accessed value need to be flushed and re-issued in the core [8, 22]. A variable hit latency also complicates cache pipelining adding to its overall complexity with special cases such as a slow-hit (a replay of a mispredicted hit).

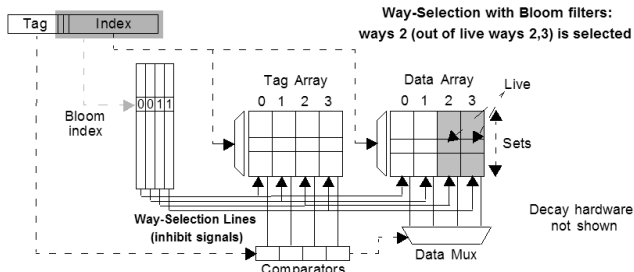
For fast scheduling of dependent instructions it is very important not only to have a fixed hit time (as we propose) but also to be able to predict misses early. Way-prediction techniques cannot predict misses — predictions are produced even on misses. Additional mechanisms to handle misses are required, e.g., predicting misses with an instruction-based predictor [26], or using Bloom filters to detect some of the misses [8]. Our approach eliminates the need to handle misses separately, encompassing the work of [8].

Some of the most sophisticated way-prediction techniques are those based on a combination of *selective direct-mapping* (DM) and way-prediction [11, 26]. In this paper we compare against one such scheme called MMRU [26]. We will describe this scheme further in Section 2.

2.2 Way-Selection

Cache decay was initially proposed to reduce leakage power in caches [14]. It is based on the generational behavior of cachelines, whereupon a cacheline initially

Fig. 3. Bloom Filters (one per way) sized to disambiguate 2 tag bits (Bloom filters not drawn to scale).



goes through a “live time”, where it is accessed frequently, followed by a “dead time”, where it simply awaits eviction. Cache decay shuts-off unneeded cachelines after some period of inactivity — called decay interval — assuming the lines have entered their dead time. The decay interval is measured with coarse 2-bit counters, local to each line, that are advanced by a global cycle counter (local counters are reset with accesses to the lines). A decay (valid) bit in the tag shows whether the line can be accessed or is dead (Fig. 2).

The simplest scheme to take advantage of cache decay for dynamic power reduction is to consult the decay status bits of the lines to enable accesses only to the live ways, and obtain dynamic power savings. We call this technique *way-selection* to distinguish it from way-prediction. The difference is that way-selection selects *zero* or more ways (up to *all* the ways) for each access whereas way-prediction strictly chooses a single way to access at all times. Because of this difference way-prediction can mispredict and require a second access, while way-selection cannot. In the case where all the lines in a set are dead, our technique predicts the miss simply by looking at the decay bits.

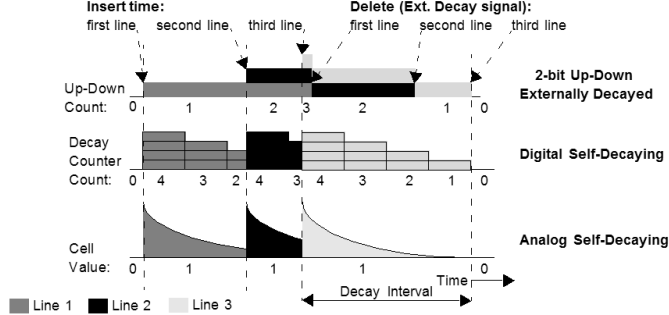
Decay bits, however, do not offer very high accuracy: many times more than one line in a set is live and it is rare to find a completely dead set (in order to predict a miss). In other words, decay bits do not disambiguate beyond the cache index. Ideally, we want to disambiguate addresses at a chosen depth. Thus, we can predict with high probability whether a *specific* line is live rather than obtain a blanket prediction of which ways are live in the line’s set. The benefits are two-fold: high probability of identifying a *single* way for a line for possible hits and high probability of *identifying misses early*.

Furthermore, decay bits are typically embedded in the tag array, while we would like to have a separate array that we can access prior to the cache, similarly to other way-prediction structures [4, 11, 26]. We accomplish our goals using appropriately sized Bloom filters [3] instead of decay bits as our predictors.

2.3 Decaying Bloom filters

Bloom filters [3] are hash tables that implement a non-membership function. Each Bloom filter entry is a single bit: 1 denotes presence of an object hashed on

Fig. 4. Behavior equivalence for 3 different implementations of a single Decaying Bloom filter entry which accomodates 3 distinct cachelines overlapping in time.



this entry; 0 denotes absence of any object that can hash on this entry. A Bloom filter tells us with certainty when something is *not present*, but it cannot tell us what exactly is present because of possible conflicts in its entries. Way-selection with Bloom filters is shown in Fig. 3 where the role of the decay bits now is performed by Bloom filters sized to disambiguate two additionaltag bits.

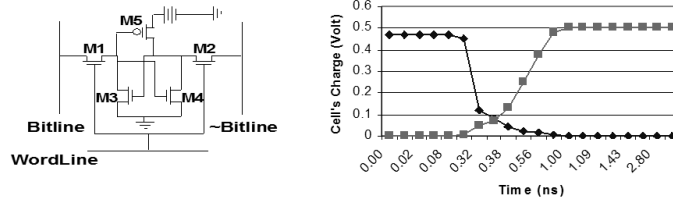
The main weakness of Bloom filters is that it is difficult to update them as there is no way to tell what is in them. In order to solve this problem, Peir et al. use helper structures called *Collision Detectors* [8] while Sethumadhavan et al. propose using small n -bit up-down saturating counters as Bloom filter entries [17]. In this case, a Bloom filter entry stores the number of objects that hash on it. As long as the total number of inserted objects does not exceed the maximum count, the Bloom filter works as expected. In contrast, we use *decay* to update our Bloom filters: deletion of a dead line from the Bloom filters is equivalent to the decay of the corresponding Bloom entry. Note that, a Bloom entry stays live until the last of the lines that hash on it decays.

Decaying Bloom filters track only live data and thus are more accurate. Their entries can decay using feedback from a decaying cache, or *autonomously*. The latter variety, which we call *Self-Decaying Bloom filters (SDBF)* is very appealing in terms of implementation simplicity since it *does not require a feedback connection to the cache*.

While feedback or externally decayed Bloom filters are implemented with 2-bit counters [17], SDBFs can be digital or analog implementations. Digital implementations use 2-bit counters corresponding to the local decay counters of the cache, so they can be quite expensive. Thus, for SDBFs, the analog design offers the least expensive implementation. We propose using decaying 4-Transistor quasi-static RAM cells [23] as Bloom filter entries. Once written, these cells lose their charge because of leakage after some preset **decay interval** unless they are accessed and, consequently, recharged.

Functionally, all the decaying Bloom filter implementations are equivalent. Figure 4 shows the behavior of a single Bloom entry on which three distinct lines are hashed. The top diagram depicts the behavior of a 2-bit saturating counter

Fig. 5. 5-T Asymmetrically Decaying Bit.



which is externally decayed. As live lines are hashed on the entry its counter is increased to three; it is decremented upon decay of the lines. The middle diagram shows the behavior of a digitally implemented SDBF. The decay counter is reset to its maximum count whenever a line is accessed and progressively decreases. Only when all lines decay the entry’s counter can fall to zero. Finally, the bottom diagram shows an analog SDBF where the behavior of the decay counter is replaced by the discharge behavior of a quasi-static memory cell.

3 Efficient Analog Implementations

Analog implementations of SDBFs being inexpensive and power-efficient, are appealing. We need, however, to carefully design them for the required functionality. This section discusses analog implementations issues and our solutions.

A decayed 4T cell yields a random value. Regardless of the value written in the 4T cell, decay means that its high node drops to the level of its low node (which floats slightly above ground at the decayed state). Thus, a decayed 4T cell is not capable in producing a significant voltage differential on its bit lines, causing the sense-amps to assume a random state at the end of their sense time. Furthermore, the cell itself might flip to a random state when connected to the bit lines. This is not a concern if the 4T stores non-architectural state [23]. But in our case, we wish to have a specific decay behavior: ones decay to zeros to signify absence of the corresponding line but zeros should remain intact since the benefit in Bloom filters comes from the certainty they provide.

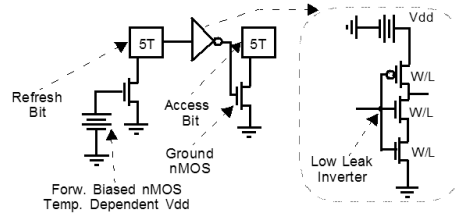
Figure 5 shows our solution. We simply re-instate one of the pMOS transistors connected to Vdd of the 6T design in the 4T design. By doing so we create an asymmetric cell, that stores logical zeroes in a static fashion and flips decayed logical ones to zeros. In Fig. 5 we can see a SPICE simulation of a typical flipping procedure. While the cell is in the flipping phase, reading its value is unpredictable. Fortunately, the window that this can happen is very small (Table 1). We handle such transient states as a soft error. Even though these reading errors might not be as infrequent as soft errors, they are still quite rare and this is why they are not a significant source of performance degradation [20].

For our 5T cells it is important to control the retention and flip times. These times greatly depend on technology: as we move to nano-scale technologies, retention times and flip times become smaller. Frequencies scale too, but in a more conservative way. We need retention times in the range of a few thousand cycles

Table 1. Retention and Flip Times for Various RAM Cells.

Type of Cell	Retention Time	Flip Time (cycles)
6T (cycle=0.2ns)	- (cycles)	-
5T (cycle=0.2ns)	1500 (cycles)	75
5T+1 (cycle=0.2ns)	1500000 (cycles)	590000
4T (cycle=0.2ns)	700 (cycles)	-
4T+1 (cycle=0.2ns)	4500 (cycles)	-

Fig. 6. Decoupled access-refresh decay cells.



but a 5T cell cannot achieve such long retention times by itself. To adjust its retention time we introduce an extra nMOS in series to ground. Forward-biasing such ground nMOS allows us to precisely control leakage currents, practically placing a resistor in the 5T cell's path to ground. Table 1 shows a typical set of retention and flip times for 4T, 5T, and 6T cells, for 70nm technology, and a 5GHz clock, with and without ground transistors. Our aim here is not to give absolute numbers but to show that we have the capability to design retention times that suit our architectural needs.

Our design also solves another problem with 4T decay, namely its dependence to temperature (leakage is exponential to temperature). An inexpensive 4T Decay Temperature sensor [15,16] monitors temperature and biases the ground transistor appropriately to yield desired decay times.

A live cell is recharged when read. This is an inherent property of the quasi-static cells and the basis of keeping accessed entries alive. However, the problem is that we inadvertently keep other Bloom-filter entries alive (which should have decayed) just by reading them. The correct refresh behavior is to refresh only the entry corresponding to a hit and nothing else. Our solution is to decouple an entry's access from its refresh. Each entry is composed of two bits: an access bit (just for reading) and a refresh bit (for updating). The refresh bit is charged only in the case of a hit in the corresponding way while the access bit is redundantly refreshed with every access. The novelty of our design is that these two bits are coupled in such a way that when the refresh bit decays, it forces the access bit to promptly follow. This is accomplished simply by connecting the high node of the refresh bit to the leakage-control ground transistor of the access bit via a low-leak inverter. When the refresh bit decays, leakage in the access bit is increased by two orders of magnitude. The detailed design is shown in Fig. 6.

Soft errors and short decay intervals. Soft-errors, 5T transient states, or thermal variations, can lead to Bloom decay errors which appear as artificially

shortened decay intervals. In this case, a Bloom filter entry decays before its corresponding cacheline. The Bloom filter reports misleadingly that the line is not in the cache. Since the benefit comes from trusting the Bloom filter and not to search the cache, in case of an error we experience a *Bloom-Induced Miss* — the line is still in the cache.

The solution to the Bloom-induced misses is readily provided in cache hierarchies that support coherency — such as those in the majority of high-performance processors today. Coherent cache hierarchies typically adhere to the inclusion property [1], e.g., L1’s contents are strictly a subset of L2’s. To support inclusion, snooping caches implement an inclusion bit alongside their other coherence bits (MESI). The inclusion bit is set if the line exists at a higher level and cleared in the opposite case. Under this scenario *the L2 does not expect a miss from the L1 for a line whose inclusion bit is set* — a sign of a Bloom-induced miss. Thus, the L2 acts as a safety net for L1 Bloom-misses. Bloom-induced misses are detected in L2 and L1 is advised of its mistake. The penalty for a Bloom-induced miss is an access of the L2’s tags. There is no other associated overhead, such as data transfer. It is evident that this safety-net protection is not offered for the L2 (the level before main memory) where the cost of a Bloom-induced miss a — main memory access — is also very high. In this case we use digital SDBFs or digital externally-decayed Bloom filters.

4 Methodology

Here, we discuss our simulation setup, details of the MMRU scheme we compare against and latency and power issues affecting the comparisons.

Simulation Setup. For our simulations we use Wattch [6] and Cacti [24]. We simulate a 4-issue processor with 32K, 32-byte block, 3-cycle L1 cache and a 1M, 64-byte block, 8-way, 15-cycle L2 cache. For the L1 we assume, optimistically, a 3-cycle pipelined design [22]. Although we do not penalize competing approaches with mis-scheduling overhead we refer the reader to [8] for its performance quantification. Our performance results do reflect, however, the negative impact of slow hit times (3+3 cycles) for way-prediction as well as the latency benefits of predicting misses (1 cycle). We run all SPEC2000 benchmarks with the reference inputs, skipping the first billion instructions and collecting statistics for the second half billion. In all graphs the benchmarks are sorted by the base case miss-rate (lowest to highest).

MMRU and Bloom filter sizing. Multi-MRU or MMRU [26] is a virtual “selective-DM” extension of the Most Recently Used (MRU) way-prediction [4]. MRU returns the most recently accessed way of a set as its prediction but MMRU allows multiple MRU predictors to disambiguate among different tags (Fig. 7). All tags in a set ending with the same bits are tracked by the same MRU table.

While we can use Bloom filters of any size — trading their accuracy — we size the Bloom filters to equal aggregate size to MMRU way-predictors. This configuration is shown in Fig. 8 for 4-way caches. The Bloom filters are shown partitioned according to the two lower tag bits for direct comparisons with the

Fig. 7. Multi-MRU employs N MRU predictors ($N == \text{assoc}$) to disambiguate among different tags.

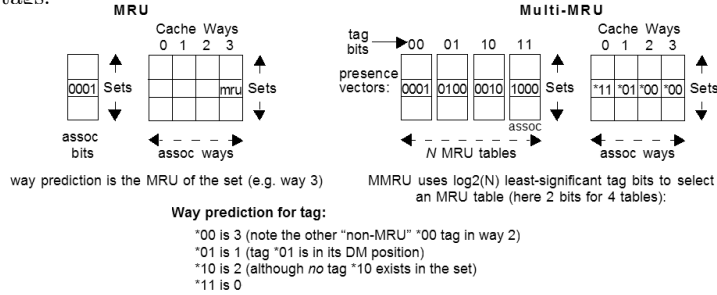
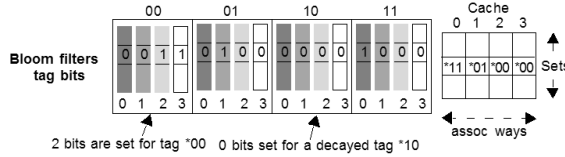


Fig. 8. 4 partitions of 4 BFs (one per way) sized to disambiguate 2 tag bits. Total size is equal to MMRU and differences (with Fig. 7) are shown for tags *00 and *10.



MMRU. Note the important differences from Fig. 7: the four Bloom filters perway collectively allow more than one bit in the presence vector for the *00 tag; the presence vector for a decayed tag (e.g., tag *10) is 0 — it is the decay of the Bloom entries that give us the ability to predict misses.

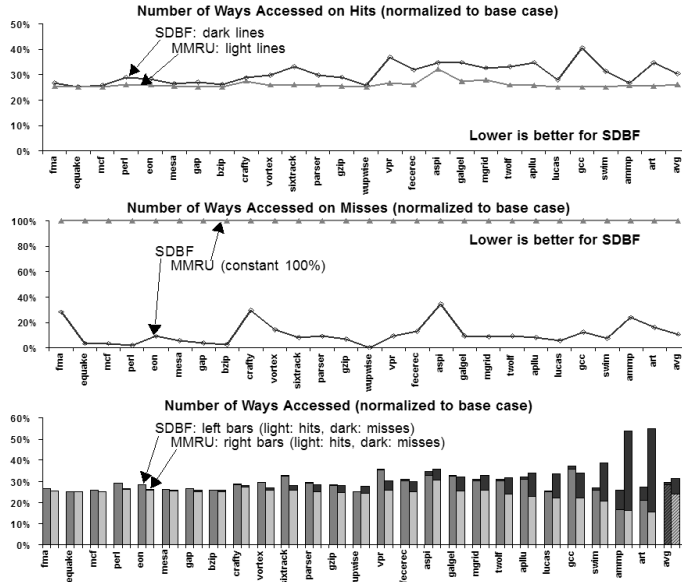
Latency issues. Early way-prediction papers focused on reducing the latency of L1 set-associative caches [2, 4, 10, 5]. For such schemes to work well, way-prediction must be performed prior to cache access. Approximate-address indexing schemes [4] or instruction-PC indexing [2, 5] were proposed for this reason. PC-based schemes suffer from low prediction accuracy and implementation complexity [11]. Consistently, using the effective address yields the highest prediction accuracy [2, 4, 5, 11].

Our view is that it is not as critical any longer to use approximate indexing schemes, given that L1 caches are not single-cycle. L1 caches are by necessity pipelined [22]. In this environment, predictor access can take place in the first pipeline stage or even occupy its own stage. Predictor access proceeds in parallel with decode and halts further operations in subsequent cycles once the predictor outcome is known. Our Cacti simulations show that accessing a small predictor structure of few kilobits is comparable in latency to L1 decode, in accordance to previous work. Power expended in cache decode, in parallel with predictor access, is fully accounted in our methodology.

Power issues. Our approach introduces two sources of additional power: i) Bloom filters arrays, and ii) decay-induced misses. Our Cacti estimates showed that energy for accessing the largest predictor (Bloom or MMRU) is less than 1% of a full L1 access (in accordance to prior work [11, 26]).

We consider the extra dynamic power of decay-induced misses (extra accesses to lower level caches due to incorrect decay of cachelines) to be part of the leakage savings [14]. Having a decaying cache to reduce leakage, one would have

Fig. 9. 4-way SDBF vs. MMRU: ways accessed on hits, on misses and in total.



to pay for this dynamic-power penalty regardless of any additional mechanism for dynamic power reduction. Of course, *we do not count leakage savings* as part of our benefits. However, we do take into account the performance penalty due to decay. As we will show, we turn this performance penalty into a performance benefit with way-selection and SDBF.

5 Results

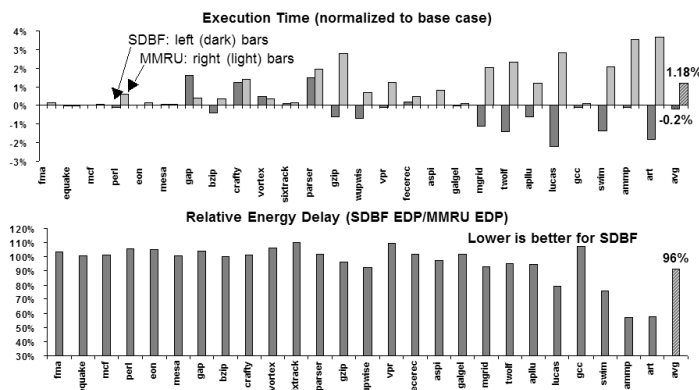
We show simulation results for two cases: first we show how decaying Bloom filters compare against MMRU for 4-way caches. We then show the scalability of our approach to higher associativity (8-way and 16-way).

5.1 Decaying Bloom filters vs. MMRU

The main result of this section is that *way-selection with Decaying Bloom filters* is more power-efficient than the highly accurate MMRU. As we show, this is mainly because our approach predicts misses.

SDBFs use a decay interval of 8Kcycles. They are coupled to a decaying cache (32KB) with the same decay interval. We use a fixed decay interval without adaptive decay [7, 14]. However, 8Kcycles is not the ideal decay interval for many of the benchmarks [21]. With SDBFs, one can adjust only a global decay interval, thus Adaptive Mode Control [7] or the Control-theoretic approach of [18] are appropriate adaptive schemes for our case but not the per-line adaptive decay [14]. We expect that adaptive decay is largely orthogonal to our work and

Fig. 10. 4-way SDBF vs. MMRU: execution time, relative EDP.



it will further improve our proposal. The most interesting results in this section, however, are about SDBF vs. MMRU. Both are equal in cost and disambiguate 2 tag bits beyond the cache index.

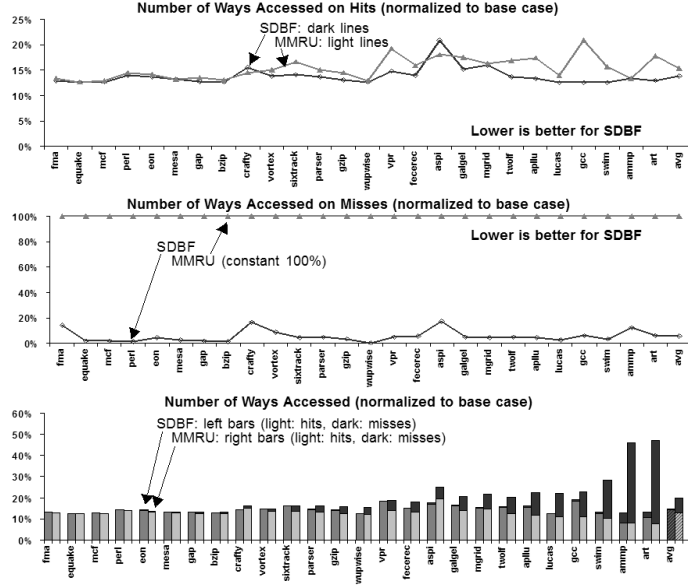
The top graph in Fig. 9 shows the number of ways accessed on hits for SDBF and MMRU. The numbers are normalized to the number of ways accessed by the base case without any prediction. SDBF lags behind MMRU — this is the result of selecting more than one way to access in our predictions. In contrast, SDBF does very well in reducing the number of ways accessed on misses (Figure 9, middle), predicting zero ways for most misses for an average of 10% of the total ways of the base case for misses. In fact, the greater the miss rate of the benchmark, the better SDBF does compared to MMRU which always accesses 100% of the ways of the base case on misses. On average SDBF accesses 29% of the total ways of the base case, while MMRU accesses 31% (Fig. 9, bottom).

In terms of execution time, SDBF speeds up many benchmarks, MMRU slows down most benchmarks (by about 1% on average, but more than 3% for the high-miss-rate ammp and art as shown in Fig. 10, top graph). The very small slowdown of MMRU attests to the fact that we do not simulate in enough detail the negative effects of variable hit latency on the cache pipeline or on the scheduling of dependent instructions [8]. On the other hand, it should be noted that SDBF speeds-up a decaying cache which is inherently at a performance disadvantage (compared to a normal non-decaying cache) because of decay-induced misses. Under these conditions the relative energy-delay product (EDP) of the SDBF is 9% better than MMRU’s (Fig. 10, bottom graph).

5.2 Scaling with Associativity

In this section we discuss how associativity affects SDBF and MMRU. Fig. 11 and Fig. 12 show the results for an 8-way 32KB L1 cache. SDBF improves dramatically in 8-way caches for both hits and misses. Its improvement in hits (Fig. 11, top graph) allows it to easily approach MMRU while further increasing its distance for the misses (Fig. 11, middle graph). Note also the improvement

Fig. 11. 8-way SDBF vs. MMRU: ways accessed on hits, on misses and in total.



in the two high-miss-rate benchmarks `ammp` and `art` because of the improved prediction of misses.

With respect to execution time (Fig. 12, top graph) for SDBF, overhead due to decay-induced misses tends to be more pronounced since the base 8-way cache has an improved miss rate. However, this is balanced by the improved prediction of misses and the resulting “fast” misses. Overall, we have no significant change for SDBF speed-up. For MMRU, we have a small increase in its slow-down (across most benchmarks) as a combined effect of decreased accuracy and improved miss-rate of the base case. In 8 ways, energy-delay is improved for MMRU but not as much as for SDBF which doubles its distance from MMRU to 21%.

Finally, Table 2 compares the averages (over all benchmarks) for the various metrics for up to 16-ways. Ways accessed are reported as a percentage of the base case without prediction. As we can see SDBF matches the performance of MMRU on hits in 16-ways while at the same time increases its distance significantly for the misses. The result is that SDBF halves its percentage of total ways accessed (29% for 4-ways, 15% for 8-ways, 7% for 16-ways) with each doubling in associativity while MMRU improves much slower (31%, 20%, 14% respectively). Power savings follow the trend of the total ways accessed and relative EDP improves with each doubling of the associativity: SDBF is 9%, 21% and 31% better than MMRU in EDP for 4-, 8- and 16-way caches.

6 Related Work

Techniques for dynamic-power reduction in caches can be divided into two categories: cache resizing and way-prediction. Cache resizing (e.g., DRI cache [19],

Fig. 12. 8-way SDBF vs. MMRU: execution time and relative EDP.

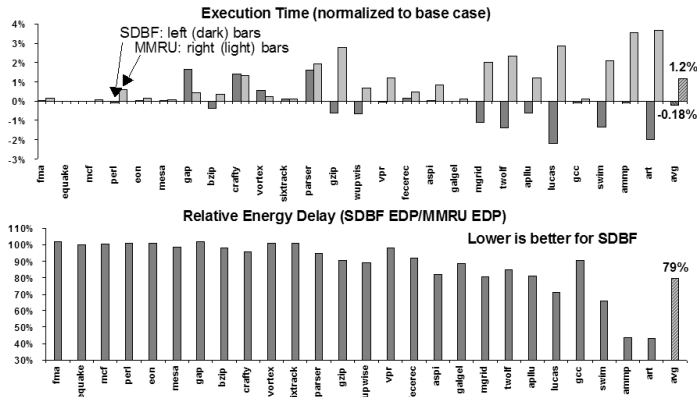


Table 2. Comparison of SDBF and MMRU for various cache associativities.

	4-ways		8-ways		16-ways	
	MMRU	SDBF	MMRU	SDBF	MMRU	SDBF
ways accessed on hits	26%	30%	14%	15%	7%	7%
ways accessed on misses	100%	10%	100%	5%	100%	2%
total ways accessed	31%	29%	20%	15%	14%	7%
power savings	59%	61%	69%	74%	75%	81%
relative EDP	91%		79%		69%	

CAM-tag resizing [12]) is a straightforward technique to reduce dynamic, static, and precharge power in caches. However, our approach can be contrasted more directly to way-prediction.

The most sophisticated way-prediction techniques are a combination of selective direct-mapping (DM) and way-prediction [11, 26]. The idea of selective direct-mapping is to assign each line to a specific way based on its lower \log_2 (associativity) tag bits. Either the line truly occupies its assigned way or this assignment is virtual and used simply to obtain predictions for this line [26]. There are two separate proposals for the combined selective-DM/way-prediction technique: by Zhu and Zhang [5, 26] and by Powell et al. [11] based on an earlier (latency-focused) paper by Batson and Vijaykumar [2].

We have already described the Zhu and Zhang proposal [26] (MMRU), a virtual “selective-DM” extension of the MRU way-prediction [4]. Zhu et al. also examine schemes where the DM assignment is actual rather than virtual but such schemes require line swapping and complex replacement policies.

The selective DM approach of Powell et al. [11] is an actual DM assignment where the lines are placed in their DM position unless they generate conflicts. In terms of prediction power this scheme aims to place as many lines as it can in their DM positions and handle the rest with a way-predictor. MMRU tracks all such lines, both those in their DM position and those in other positions, thus

yielding approximately the same prediction accuracy textbf— for both schemes, on average 92% of the predictions result in first hits for 4-way caches [11, 26]. Because of this prediction equivalence we compare our approach to MMRU noting that our conclusions would be similar with respect to the Powell approach.

Finally, way-selection is similar to the way-halting cache proposed by Zhang et al. [25] where accesses to ways that cannot possibly contain the desired line are “halted”. Way-halting is limited by the use of expensive CAMs for the lower four bits of the tags to determine tag mismatch per way and thus difficult to scale to a large number of sets. In contrast, we use Bloom filters instead of CAM-based halt tags, thus providing better scalability. Our decay techniques, however, can benefit way-halting too, clearing the halt tags from useless entries and increasing their effectiveness in predicting misses.

7 Conclusions

In this paper, we propose a dynamic power application for a leakage-power mechanism, allowing the seamless integration of the two. To achieve dynamic power reduction with decay we propose way-selection (similar to way-halting) and Decaying Bloom filters. Way-selection offers a significant advantage over way-prediction: that of a fixed hit time. We also propose several variations of Decaying Bloom filters, most notably the analog Self-Decaying Bloom filters, that decouple decay in our predictors from decay in the cache. This gives us the flexibility to use Decaying Bloom filters with non-decaying caches (or for instance state-preserving Drowsy caches). Decay, however, is necessary for the Bloom filters to be efficient. Another advantage of our approach is that it integrates prediction of misses. We have shown that our approaches can outperform some of the most sophisticated way-prediction proposals. And this, without taking into account leakage savings of the integrated dynamic/leakage power mechanism.

Acknowledgments

This work is supported by the HiPEAC Network of Excellence, the European SARC project No.027648 and by Intel Research Equipment Grant #15842.

References

1. J.-L. Baer and W. Wang. On the inclusion properties for multi-level cache hierarchies. In *Proc. of the Int. Symp. of Computer Architecture*, 1988.
2. B. Batson and T. N. Vijaykumar. Reactive-associative caches. In *Proc. of PACT*, 2001.
3. B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7), 1970.
4. B. Calder et al. Predictive sequential associative cache. In *Proc. of the Symp. on High-Performance Computer Architecture*, 1996.

5. C. Zhang et al. Two fast and high-associativity cache schemes. *IEEE Micro*, 17(5), 1997.
6. D. Brooks et al. Wattach: a framework for architectural-level power analysis and optimizations. In *Proc. of the Int. Symp. of Computer Architecture*, 2000.
7. H. Zhou et al. Adaptive mode control: A static-power-efficient cache design. *Trans. on Embedded Computing Sys.*, 2(3), 2003.
8. J.-K. Peir et al. Bloom filtering cache misses for accurate data speculation and prefetching. In *Proc. of the Int. Conference on Supercomputing*, 2002.
9. K. Flautner et al. Drowsy caches: Simple techniques for reducing leakage power. In *Proc. of the Int. Symp. of Computer Architecture*, 2002.
10. K. Inoue et al. Way-predicting set-associative cache for high performance and low energy consumption. In *Proc. of ISLPED*, 1999.
11. M. Powell et al. Reducing set-associative cache energy via way-prediction and selective direct-mapping. In *Proc. of the Int. Symp. on Microarchitecture*, 2001.
12. M. Zhang et al. Fine-grain cam-tag cache resizing using miss tags. In *Proc. of ISLPED*, 2002.
13. R. Min et al. Location cache: a low-power l2 cache system. In *Proc. of ISLPED*, 2004.
14. S. Kaxiras et al. Cache decay: exploiting generational behavior to reduce cache leakage power. In *Proc. of the Int. Symp. of Computer Architecture*, 2001.
15. S. Kaxiras et al. 4t-decay sensors: a new class of small, fast, robust, and low-power, temperature/leakage sensors. In *Proc. of ISLPED*, 2004.
16. S. Kaxiras et al. A simple mechanism to adapt leakage-control policies to temperature. In *Proc. of ISLPED*, 2005.
17. S. Sethumadhavan et al. Scalable hardware memory disambiguation for high-ilp processors. *IEEE Micro*, 24(6), 2004.
18. S. Velusamy et al. Adaptive cache decay using formal feedback control. In *Proc. of the Workshop on Memory Performance Issues.*, 2002.
19. S. Yang et al. An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance i-caches. In *Proc. of the Int. Symp. on High-Performance Computer Architecture*, 2001.
20. V. Degalahal et al. Analyzing soft errors in leakage optimized sram design. In *Proc. of the Int. Conference on VLSI Design*, 2003.
21. Y. Li et al. State-preserving vs. non-state-preserving leakage control in caches. In *Proc. of the Conference on Design, Automation and Test in Europe*, 2004.
22. Z. Chishti et al. Wire delay is not a problem for smt (in the near future). In *Proc. of the Int. Symp. of Computer Architecture*, 2004.
23. Z. Hu et al. Managing leakage for transient data: decay and quasi-static 4t memory cells. In *Proc. of ISLPED*, 2002.
24. S. Wilton and N. Jouppi. Cacti: An enhanced cache access and cycle time model. In *IEEE Journal of Solid-State Circuits*, Vol. 31(5):677-688, 1996.
25. C. Zhang and K. Asanovic. A way-halting cache for low-energy high-performance systems. *ACM Trans. Archit. Code Optim.*, 2(1), 2005.
26. Z. Zhu and X. Zhang. Access-mode predictions for low-power cache design. *IEEE Micro*, 22(2), 2002.