ARI: Adaptive LLC-Memory Traffic Management

VIACHESLAV V. FEDOROV, Texas A&M University
SHENG QIU, Texas A&M University
A. L. NARASIMHA REDDY, Texas A&M University
PAUL V. GRATZ, Texas A&M University

Decreasing the traffic from CPU LLC to main memory is a very important issue in modern systems. Recent work focuses on cache misses, overlooking the impact of writebacks on the total memory traffic, energy consumption, IPC, etc. Policies which foster a balanced approach, between reducing write traffic to memory and improving miss-rates can increase overall performance, improve energy efficiency and memory system lifetime for NVM memory technology, such as Phase-Change Memory (PCM). We propose Adaptive Replacement and Insertion (ARI), an adaptive approach to last level CPU cache management, optimizing the two parameters (miss rate and writeback rate) simultaneously. Our specific focus is to reduce writebacks as much as possible while maintaining or improving miss rate relative to conventional LRU replacement policy. ARI reduces LLC writebacks by 33% on an average while also decreasing misses by 4.7% on an average. In a typical system, this boosts IPC by 4.9% on an average while decreasing energy consumption by 8.9%. These results are achieved with minimal hardware overheads.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—cache memories; C.0 [**General**]: Modeling of Computer Architecture

General Terms: Design, Cache, Memory, Performance, Simulation

Additional Key Words and Phrases: LLC, insertion policy, replacement policy, main memory, bandwidth, sampling

ACM Reference Format:
Viacheslav V. Fedorov, Sheng Qiu, A. L. Narasimha Reddy, and Paul V. Gratz. 2013. ARI: Adaptive LLC-Memory Traffic Management. *ACM Trans. Architec. Code Optim.* 0, 0, Article 0 (January 2013), 20 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

The working sets of modern applications keep increasing, which in turn is placing greater pressure on the memory system, both in terms of performance and capacity. At the same time, power and energy constraints in current process technologies are imposing new limits on off-chip bandwidth. Furthermore, traditional DRAM-based main memory is now hitting hard power, performance and capacity constraints that will limit process technology scaling [International Technology Roadmap for Semiconductors (ITRS) Working Group 2009]. Alternate memory technologies, such as Phase-Change Memory (PCM), have been proposed to alleviate main memory capacity and scalability concerns, however these technologies impose further costs on off-chip writes in terms of power consumption and wear-out. In this article we present an adaptive approach,

This research is supported in part by the National Science Foundation, under grant CCF-1320074.
Authors' addresses: V. V. Fedorov, S. Qiu, A. L. N. Reddy, and P. V. Gratz, Computer Engineering and Systems Group, Department of Electrical and Computer Engineering, Texas A&M University.

DOI:http://dx.doi.org/10.1145/0000000.0000000

leveraging program phase behavior, to address all of these challenges. In particular, we propose to adaptively modify Last-level Cache (LLC) management policy to simultaneously reduce writebacks while improving miss-rate, addressing both power and performance concerns. Although there has been considerable prior work exploring LLC management policies to reduce miss-rate and improve performance and some prior work examining policies to reduce writebacks for optimized bandwidth consumption, we present the first work to our knowledge to simultaneously address both.

To mitigate the greater memory system pressure placed by applications on their memory systems to maintain data and instruction stream needs, current chips employ memory system hierarchies with several levels of cache prior to main memory [Molka et al. 2009]. Last-level Caches (LLCs) are optimized towards capacity rather than speed, and are often highly associative. LLCs typically employ Least Recently Used (LRU) or approximations of LRU policies to choose which victim block is to be replaced when a cache miss occurs. If the victim block is dirty, it must be written to memory before the new incoming block may be written to cache. These writebacks constitute an increase in off-chip bandwidth consumption, particularly for cache-lines that ping-pong back and forth between the LLC and main memory. Furthermore, for alternate memory technologies such as PCM, this writeback cost is particularly high and undesirable.

Considering the high costs of writebacks, it is better to replace a clean cache block in the LRU stack rather than the dirty LRU one, however, the replaced clean cache block should not generate more misses otherwise overall system performance might suffer. We observed that the number of hits often distributes unevenly among cache ways of the LRU stack in the LLC (see Figure 1). Most of hits accumulate on the first few MRU ways, with a much lower, flat distribution among other parts of the LRU stack. Based on this observation, we can avoid generating writeback traffic by replacing those clean blocks in the LRU stack which have less (or comparable) hits relative to the dirty LRU ones without introducing additional misses.

In exploring LLC hit distribution across applications, we find they vary greatly from application to application and even from program phase to program phase within a given application. A one-size-fits-all approach to LLC management either imposes a significant cost on miss-rate for some applications or does not reduce writebacks sufficiently. In this article we propose an LLC management policy which adapts to program phases such that both writebacks and miss-rate are improved. The primary contributions of this work are as follows:

— An adaptive cache management scheme, ARI, that simultaneously reduces *both* the miss-rate and writeback rate compared to LRU, to accommodate the behavior of different applications and different phases of an application.
— Our design reduces main memory energy consumption, while increasing IPC by 4.9% on average over traditional LLC cache design.
— When used in conjunction with PCM-based main memory, ARI improves system lifetime on average by 49% beyond a randomized wear-leveling baseline [Qureshi et al. 2009a; Seong et al. 2010].

The remainder of the article is organized as follows: In Sections 2 and 3 we provide design and implementation details on ARI which is then evaluated in Section 4. We present further analysis on our design in Section 5. We then discuss prior work in cache replacement and PCM-based main memory in Section 6. Section 7 concludes the article.

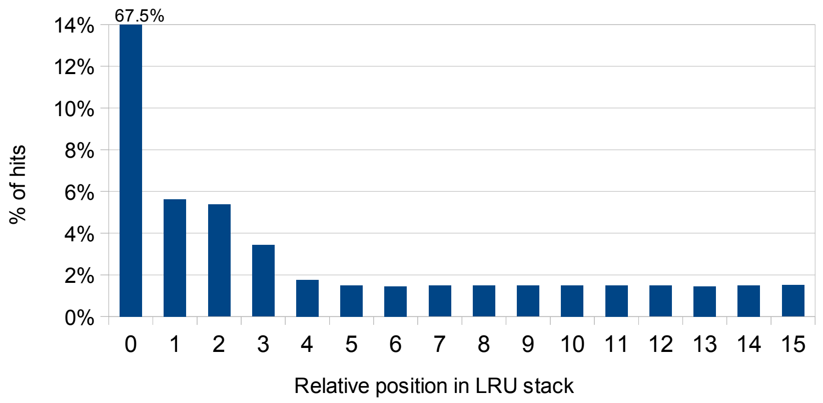


Fig. 1: Distribution of total number of hits across a 16-way, 2MB LLC for *mcf* application (0 - MRU, 15 - LRU).

2. DESIGN

Our **goal** is to reduce the writebacks as much as possible, while simultaneously keeping the miss rate at least equal or better than conventional LRU policy, to avoid the performance loss. Since our scheme has comparable or better miss rate than LRU as well as writeback reduction, it produces a significant main memory bandwidth reduction and improves the system performance. Therefore, the proposed scheme, Adaptive Replacement and Insertion (ARI) is potentially useful for a broad range of future memory systems, including DRAM-only systems, and hybrid memory systems employing both PCM and DRAM.

In this section, we discuss static replacement policies which favor writeback reduction, followed by our adaptive replacement and adaptive insertion techniques which together form the proposed ARI LLC cache management policy.

As the cache hit distribution across ways in the LRU stack is nearly bimodal, we mentally divide the LRU stack into two partitions. We call these partitions the "High-hit" and "Low-hit" partitions, reflecting the number of hits in cache ways belonging to a given partition. To distinguish between different applications' varying Stack Distances, we refer to the partitioning as nHm where n is the number of cache ways belonging to the High-hit partition and m is the cache associativity (i.e., Figure 1 gives an example of a 4H16 partitioning: *mcf* benchmark has the majority of its cache hits in the four MRU ways, whereas twelve other ways receive a relatively low number of hits).

The items in the High-hit partition should obviously stay in the cache as long as possible since they are being frequently accessed. On the other hand, the items in Low-hit partition are accessed much less frequently and, importantly, *at nearly the same rate*, thus evicting a line in any position in the Low-hit partition has approximately the same effect on miss rate as any other. Therefore, evicting any clean block within the Low-hit partition is more beneficial than evicting a dirty LRU block (reducing writebacks). Thus upon a cache miss, we use a policy within the Low-hit partition which first tries to find a least recently used clean block. As an example, Figure 2 depicts potential eviction candidates within one cache set, for various hit distributions within the set (cf. Figure 1). In the case there are no clean blocks, the least recently used dirty block is replaced. If there are no Low-hit blocks (i.e., 16H16), the block in LRU position is evicted. By doing this, we greatly reduce the writeback traffic to memory without degrading overall performance of the system.

In our initial experiments we found that any given static policy, while decreasing writeback rate, will typically sacrifice miss rate relative to LRU on average across a suite of benchmarks. This is because it is difficult, if not impossible, to statically tune a single one-size-fits-all policy since different applications exhibit varying Stack Distance distribution. Furthermore, applications have execution phases where the cache accesses follow different patterns, hence the Stack Distance distribution and optimal cache partitioning varies by phase. As a result, in some applications the miss rate increases by up to 3x relative to LRU, while the writeback count is only marginally decreased when a static policy is used.

2.1. Adaptive Replacement

We propose to adapt to application behavior, as well as to the execution phases inside applications by extracting run-time information about the program execution from the cache itself. In particular, we propose sampling a small number of cache sets to estimate the current application behavior under different partitionings of the cache and choose the best cache partitioning according to that data, balancing miss rate against writeback benefit.

While it is possible to extract the stack distance distribution directly from cache and use this information to adapt the cache policy, the stack distance is a function of the replacement policy and is therefore effected by the policy currently in use. Instead, to account for the impact of the policy on cache metrics, we plan to adopt a direct measurement approach, implementing various sample partition schemes on selected sets, using shadow tags, rather than trying to measure the stack distance of the cache as a whole to guide the policy.

Our approach employs p static sample partition policies, where each policy partitions the ways in a set at a particular stack distance. Among our sampled policies, we always include two extremes, one being LRU and the other 0H16. The other policies partition their sets at different locations. To improve the accuracy of the modeling, we implement the sample policies in shadow tags, i.e. for a sampled cache set, we have additional p tag sets which implement the p static policies.

For each sampled policy i, where $i = 1...p$, we maintain two performance counters. The first counter measures the number of misses, the second counter measures the number of writebacks. These performance counters are used to compare the different sample policies at the end of an epoch. Based on the comparison of the performance counters, a policy for the entire cache is chosen for the next epoch. Rather then resetting the counters to zero at the end of each epoch, we adopt a decaying average similar to that used in RTT calculation in TCP/IP protocol.

The cache controller maintains an epoch countdown timer which counts down the epoch length before triggering the decision logic. Once the epoch has ended, the miss counts, mr(i), and writeback counts, wb(i), are summed together for each of the sampled sets, and the policy(i) with the lowest sum is chosen.

The intuition for considering the sum of mr(i) and wb(i) is that it provides a rough guideline for the savings in memory bandwidth relative to LRU, if this policy is adopted. If a policy provides a pronounced decrease in writeback rate, it may be tolerable to allow some increase in miss rate, still keeping the memory traffic lower than LRU, and vice versa. If all the sampled policies perform the same as LRU (or there are no accesses to the sampled sets), the main cache policy is not changed. After the decision hardware has picked the best policy for the current epoch, the cache controller applies it toward the whole cache.

We also tried varying the weights of mr(i) and wb(i) in the sum, to see if this could lead to a better trade-off. Our results indicate that writebacks can be further reduced, by up to 2%, when misses are not considered. Similarly, misses can be reduced by

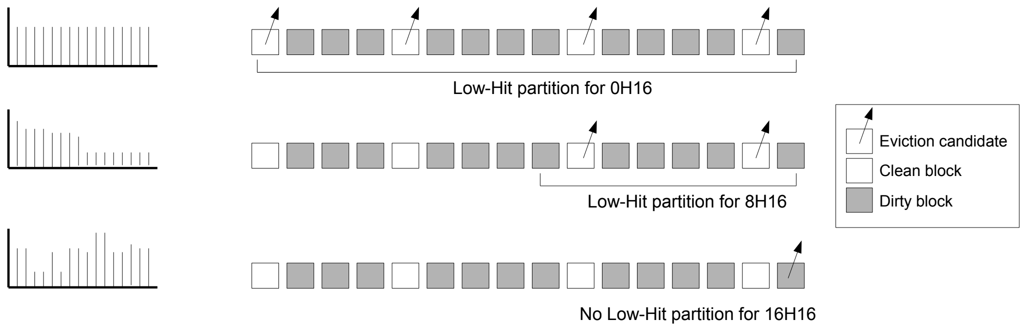

Low-Hit partition for 0H16
Low-Hit partition for 8H16
No Low-Hit partition for 16H16
Eviction candidate
Clean block
Dirty block

Fig. 2: Eviction candidates for various static policies.

up to 2% if writebacks are ignored. The equal weights used in the paper yield the best writeback improvement without negatively affecting performance.

We note that the power of sampling lies in its simplicity. More intelligent schemes might yield a better low- vs. high-hit partitioning, but would require more information than what can be extracted just from the LLC. IPC counts or an address of current load/store instruction are the examples. LLC is typically very distant from the main core, so providing such additional information to the cache controller would not only complicate the controller, but also disturb the core (i.e., stronger drivers are needed to support long wires, etc.).

2.2. Dynamic Insertion

In conventional replacement policies, when a block is brought into the cache, it is typically inserted as an MRU element in the stack; however, prior work has shown that this is not optimal for miss rate in the presence of reference streams with varying amounts of locality [Qureshi et al. 2007]. Earlier work used set dueling to choose the insertion point between the MRU or LRU position, depending on the expected locality of the inserted cache line. We also propose to adopt a dynamic insertion policy, however, we leverage our characterization of the LRU stack into low-hit and high-hit partitions to better place low locality cache blocks (effectively yielding $p + 1$ insertion locations instead of just 2 as we will show). Set dueling utilizes a single bimodal misses counter, and picks one of the two policies to use based on that counter. If the number of misses for both policies is similar, the counter might get stuck, or it might oscillate around the threshold, switching the policies even if it actually were beneficial to keep the certain policy. ARI uses miss- and writeback counts in making decisions for a number of policies for every epoch, which allows a faster and more precise adaptation. The phases in set-dueling are those when the miss rate of one policy gets sufficiently better/worse than the other as to bias the counter and trigger the policy change (i.e. if one policy initially experiences a lot of misses and then performs better compared to the other policy, it may not be chosen until the bimodal counter gets to the threshold). Phases in our work are when the application's high-hit and low-hit way distribution changes, so the appropriate policy is chosen as soon as possible.

We propose inserting clean data with expected low locality at the top of the low-hit partition, as illustrated in Figure 2, under the expectation that this will yield a lower miss rate than LRU insertion in the event our speculation is wrong. Clean data with expected high locality is always inserted in the MRU location. The intuition is, if data has low locality and we correctly insert it in the low-hit partition, we are likely to see a reduction in writebacks and misses as we don't disturb the elements belonging to the high-hit partition that are receiving significantly more hits.

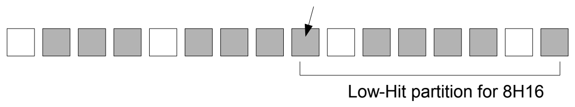


Fig. 3: Insertion point for 8H16 static policy, for low-locality blocks.

Table I: Baseline Cache Configurations

System	Single core
L1 cache	32KB L1I + 64KB L1D, 2-way, LRU, 64B block
L2 cache	256KB, 8-way, LRU, 64B block
L3 cache	2MB, 16-way, LRU, 64B block
Main memory	4GB, DDR3-1333 DRAM, 32-entry write buffer

System	Multicore
L1 cache (Private)	64KB L1I + 64KB L1D, 2-way, LRU, 64B block
L2 cache (Private)	256KB, 8-way, LRU, 64B block
L3 cache (Shared)	16MB, 16-way, LRU, 64B block
Main memory	4GB, DDR3-1333 DRAM, 32-entry write buffer

Again, we take a measurement approach to decide which insertion location reduces writebacks and misses best for the current phase of the running application. For each partitioning policy, we consider insertion at both MRU and the top of the assumed low-hit partition, and choose the insertion policy for the next epoch based on the observed behavior in the current epoch. As a result, a direct implementation of both the replacement and insertion policies, we need to double the number of shadow tags and associated counters. We will show later that this overhead can be reduced to a smaller number of tags while keeping the performance nearly the same as a full-scale implementation. We note that ARI may be used to control non-stack-based cache management policies, provided that the sample sets employ LRU to estimate the high- vs low-hit distribution in the LLC.

3. IMPLEMENTATION DETAILS

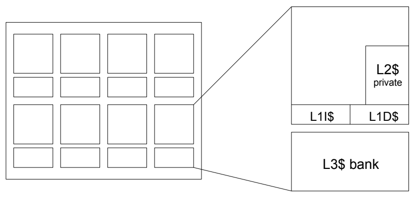
The proposed scheme is relatively easy to implement; it requires little additional hardware, and the decision-making delay is insignificant relative to our epoch length.

Figure 4a shows the baseline ARI design implemented in a 8-node CMP with three levels of cache hierarchy. L1 and L2 caches are assumed private to each core, with each node having a single bank of shared L3 cache. Our scheme assumes that the LLC is non-inclusive. Although inclusivity simplifies cache coherency, prior work has explored cache coherency in non-inclusive caches [Zheng et al. 2004; Jaleel et al. 2010a; Gaur et al. 2011].

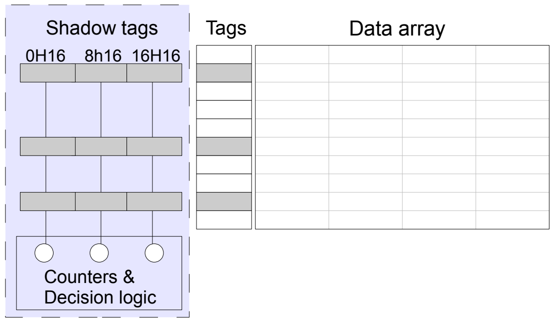
We randomly pick 8 of each L3 cache bank's sets to shadow. For each selected set we create *p*2* (*p* partition policies times two insertion policies) shadow tag sets to be sampled. Each of the *p* shadow tags for a given set implements one of the sample partitioning policies. These sample sets are doubled to include one sample set for each insertion policy: at the top of the low-hit partition (LH), and standard MRU. For example, given $p = 3$, 8 sets of sample tags will be maintained for 0H16-MRU, 0H16-LH, 8H16-MRU, 8H16-LH, 16H16-MRU (traditional LRU), and 16H16-LH per L3 cache bank.

Assuming a 16MB (2MB per core), 16-way L3 with 64-byte lines, there are 16384 total sets in the cache. The storage overhead for ARI is thus 1152, 640, and 384 tag sets, or 7%, 4%, and 2% of L3 tag array (14kB, 7.5kB, and 4.6kB of storage, using the hashed tags approach, as discussed later) with $p = 9$, 5, and 3, respectively. As we will show later, these overheads can be lowered without impacting the performance.

In a single-core configuration with a 2MB LLC, ARI induces 7kB of storage overhead; in comparison, sampling Dead-block predictor (DBLK) [Khan et al. 2010] has 13kB



(a) Baseline CMP



(b) L3 bank with shadow tags array

Fig. 4: Block diagrams of the proposed design

storage overhead for a 2MB LLC, and requires additional communication circuitry from CPU core to the LLC.

We note that no modification to the main cache memory structures is needed. Only the cache controller needs to be modified. The shadow sample sets are collocated with the L3 cache banks that contain the sets they shadow such that the addresses matching the shadow sets are sent to both the set itself and the shadow (Figure 4b). Shadow tags are independent of the main cache structures and do not affect the cache operation.

On an epoch boundary the cache controller examines the sample sets to determine the best policy. The epoch timer counts up to a maximum of 25 thousand LLC references and thus requires 15 bits. The performance counters for each sample policy can be made 15-bits wide, to ensure no counter overflows. Basing the epoch on reference count rather than cycle count allows adaptation of the epoch's resolution to the relative activity of the memory system at that time. The epoch length was determined experimentally. Once the epoch size has been set, no further tuning is necessary, as ARI dynamically adapts to the runtime phases regardless of the nature of the phases (e.g., one application using the cache in a single-core CPU, or several applications competing for multi-bank cache).

To increase the accuracy, we maintain some history in the performance counters. At the end of each epoch, we compute the performance measure as both a function of the previous history and the current sample obtained during the current epoch. This smooths the sampled data and allows better adaptation. We compute each sample according to the following formula: $new\_value = 0.9375 * previous\_value + 0.0625 * current\_value$. The fractions are chosen for ease of implementation.

To compute the miss and writeback sums, we propose using a low-power, pipelined adder. We expect the time to compute the sums and comparisons and change policy should be on the order of 100-200 cycles, insignificant relative to epoch size. To ensure we never use policies that increase misses excessively, we remove from consideration policies with positive miss deltas above the preset threshold. If the threshold is chosen carefully, (say as 6.25% or 1/16), then this threshold check can be performed with simple shift operations.

4. EVALUATION

In this section we examine the performance of ARI under various workloads. We also compare ARI to the LLC management schemes from previous work.

Table II: DRAM and PCM characteristics for 1GB chip

Power	DRAM	PCM
Row read	210 mW	78 mW
Row write	195 mW	773 mW
Activate	75 mW	25 mW
Standby	90 mW	45 mW
Refresh	4 mW	0 mW

Latency	DRAM	PCM
Initial row read	15 ns	28 ns
Row write	22 ns	150 ns
Same row read/write	15 ns	15 ns

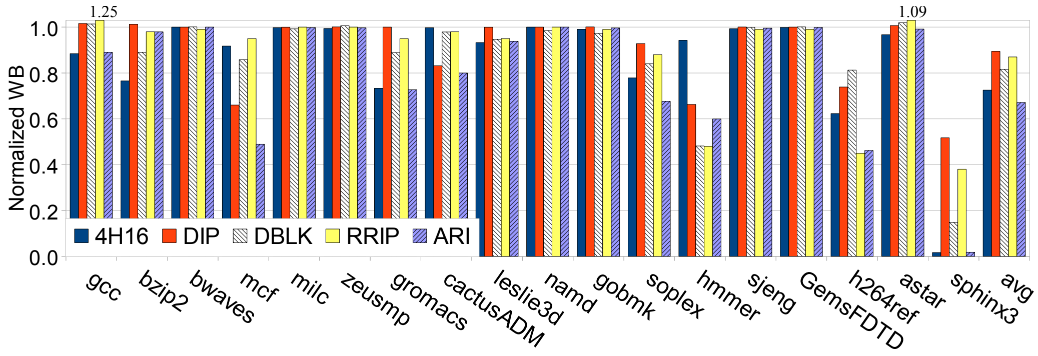


Fig. 5: Writebacks improvement, normalized to LRU.

4.1. Methodology

For single-core as well as single-processor multitasking simulations we use the gem5 simulator [Binkert et al. 2011] paired with the DRAMSim2 main memory simulator [Rosenfeld et al. 2011], running the compiled code from SPEC CPU2006 benchmark suite utilizing the single SimPoint methodology [Perelman et al. 2003]. We picked applications from SPEC 2006 package that provide good representation of the whole suite in terms of stack distance behavior and cache demands.

For multi-core and multi-threaded simulations, the gem5 simulator running the PARSEC suite is utilized. We used nVidia Tegra-like system with a dual-issue out-of-order processor, as the baseline for single-core benchmarks, and a multiprocessor system for multicore benchmarks. The cache configurations used are shown in Table I. The implementation only impacts the L3 cache, the LLC of the system. We simulated three levels of caches, but all the techniques presented in this article can be applied to any last level cache.

The latency and power estimates for DRAM and PCM shown in Table II are adopted from Dhiman et al. [2009].

4.2. Performance

In single-core configuration, ARI samples $k = 9$ distinct policies (including 0H16 and LRU) evenly distributed over the range of possible stack distances, with 32 sets for each sample policy. Figures 5 and 6 present the miss- and writeback gains for ARI as well as 4H16 static policy, DIP [Qureshi et al. 2007], Dead Block Prediction (DBLK) [Khan et al. 2010], and DRRIP [Jaleel et al. 2010b]. The results are normalized to LRU.

We see that ARI performs well, achieving 33% LLC writeback improvement on an average. ARI never causes misses to increase by more than 5% (milc, sjeng), and on an average improves them by 4.7%. In comparison, DIP, DBLK, and RRIP decrease writebacks on an average by 11%, 18%, and 13%, respectively. DIP improves misses

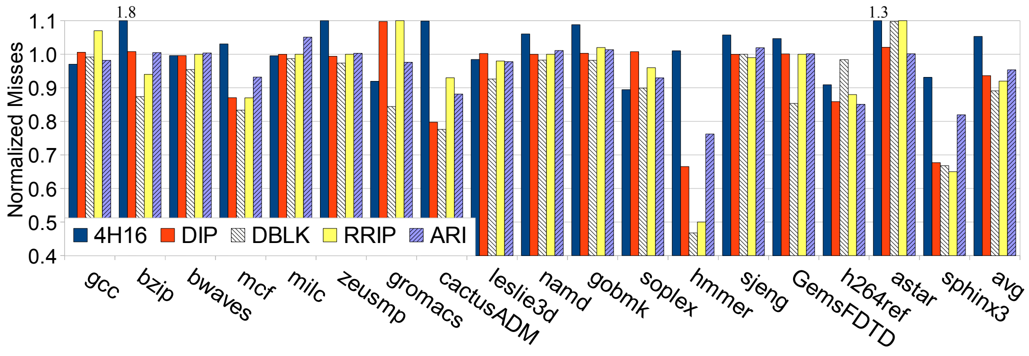


Fig. 6: Misses improvement, normalized to LRU.

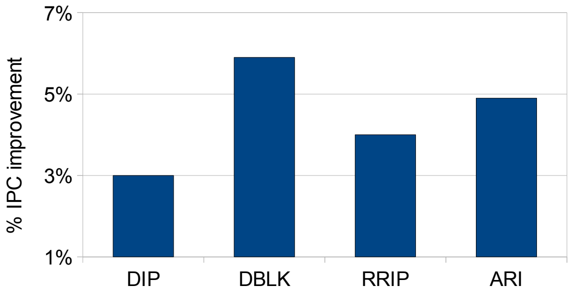


Fig. 7: IPC improvement with ARI, DIP, and DBLK, normalized to LRU.

by 6%, DBLK by 11%, and RRIP by 8%, on an average. Note how for astar application, ARI identifies the fact that there is little gain to be exploited and reverts to LRU, while DBLK and RRIP increase misses by 10%. The 4H16 static policy achieves 25% writeback improvement but increases misses by 5% on an average. Note that for bzip2 and astar 4H16 miss rate is 80% and 30% greater than with LRU. Our simulations show that as we go from 0H16 toward 16H16, the negative effect on misses declines, but so does the writeback improvement. ARI dynamically tunes the replacement policy to achieve significant improvement in writebacks and misses.

4.3. Memory Bandwidth reduction

Decreasing the traffic from CPU LLC to main memory is important in modern systems. ARI yields the main memory bandwidth reduction of up to 39% (for h264ref), and 8.3% on an average. In comparison, our simulations show that DIP, aimed at reducing LLC miss traffic, reduces the main memory bandwidth by 4.6%. RRIP decreases the bandwidth by 7.4%. Sampling dead block prediction scheme achieves 10% bandwidth reduction, although with twice the storage overhead.

4.4. Application speedup

The improvement in misses and writebacks, and thus the decreased main memory traffic, leads to IPC improvement and program speedup. Note that IPC is mostly dependent on cache misses reduction. We conclude that the writebacks are absorbed in the write buffer. They only affect the performance when write traffic congests the memory bus, or when the write buffer gets full and the CPU core is forced to wait for a write to commit. ARI decreases the number of writebacks substantially, so there is less congestion on the bus, and more opportunities to free the entries in the write buffer for the incoming blocks.

Figure 7 presents IPC improvement over the SPEC applications simulated, normalized to baseline LRU. ARI achieves a 4.9% speedup on an average, outperforming DIP and RRIP, and nearly equivalent to DBLK.

4.5. Energy and Lifetime

In order to understand the impact of reduced writebacks and misses on the memory system, we simulated three different memory architectures. The first consists entirely of DRAM, the second consists entirely of PCM, and the third employs a 256MB DRAM cache [Qureshi et al. 2009b] in front of PCM.

Note that we do not include the energy overhead introduced by ARI sampling structures in LLC, since it is negligible compared to the main memory power consumption. We estimate that ARI structures use less than 5% of the total L3 power. Taking into account that a typical LLC consumes 2.75W peak power [Khan et al. 2010], ARI adds less than 150mW of power overhead - this is half of 1GB DRAM bank read power. Furthermore, to be completely fair in reporting total system power, we would have to analyze the reduction in memory bus power, power savings due to less cache misses and faster application runtime, etc., in addition to sampling overhead. This is a matter of a separate article.

With DRAM-only memory, the total energy is mostly dependent upon application running time, as write energy consumption is not significant compared to standby and refresh energy. The average energy savings in a system employing ARI are thus 4.9%.

Because ARI is aimed at reducing writebacks, more prominent energy savings can be achieved when memory technologies with expensive write accesses, such as PCM, are employed. Applications such as mcf, h264ref and hmmer experience more than 10% savings. The average energy savings across the simulated applications are 8.9% compared to the baseline LRU.

As the baseline for PCM lifetime comparison, we use randomized wear leveling, where the cells receive equal number of writes, and thus lifetime is essentially the number of writes a cell can sustain until it is burned out. The simulations suggest that PCM-only system with ARI-managed LLC sees an average lifetime improvement of 49%, with some workloads attaining up to 2.5x lifetime extension. In contrast, DIP yields 12%, DBLK 23%, and RRIP a 15% PCM lifetime improvement on an average. However, in a system implementing DBLK or RRIP, PCM lifetime *decreases* by a negligible amount for gcc, zeusmp and astar applications.

With DRAM cache in front of PCM, the cache absorbs all the memory read and write requests (with the exception of bwaves, GemsFDTD, and zeusmp). The energy reduction is limited to 0.96% on an average, because the 256MB DRAM cache uses much less energy than 4GB DRAM memory. In low power systems, e.g. phone and tablet systems, however, DRAM caches may be too expensive.

4.6. Multiprocessors

We ran nine PARSEC applications to explore ARI's performance in a multiprocessor configuration with a 16MB LLC. The results are presented in Figures 8 and 9 below, with the rates normalized to baseline LRU. The last three bars are geometric mean over the 9 applications. Once again, we observed that ARI does not increase miss rate by more than 4% for any single application, while reducing the writeback rate by 20% on an average and keeping the average miss rate same as LRU policy. In contrast, the 4H16 static policy reduces writebacks by 29% on an average, but allows miss rate to increase by 26% in the worst case, and by 9% on an average.

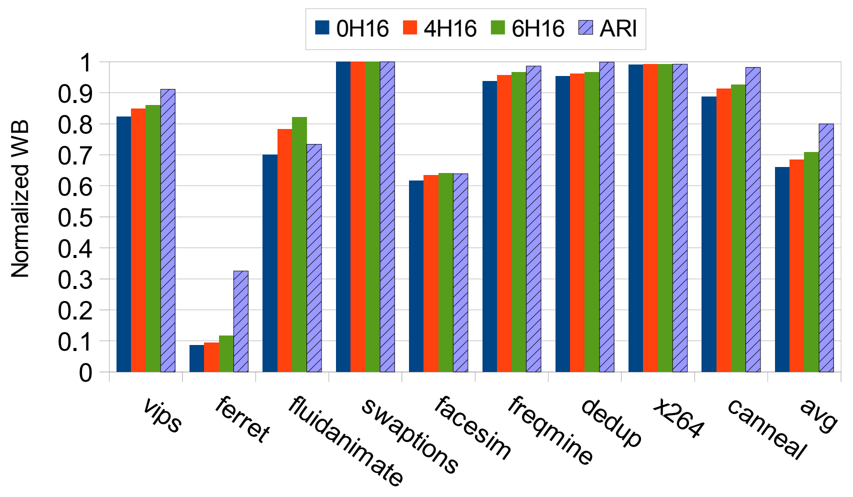


Fig. 8: Writebacks for the PARSEC applications, normalized to baseline LRU.

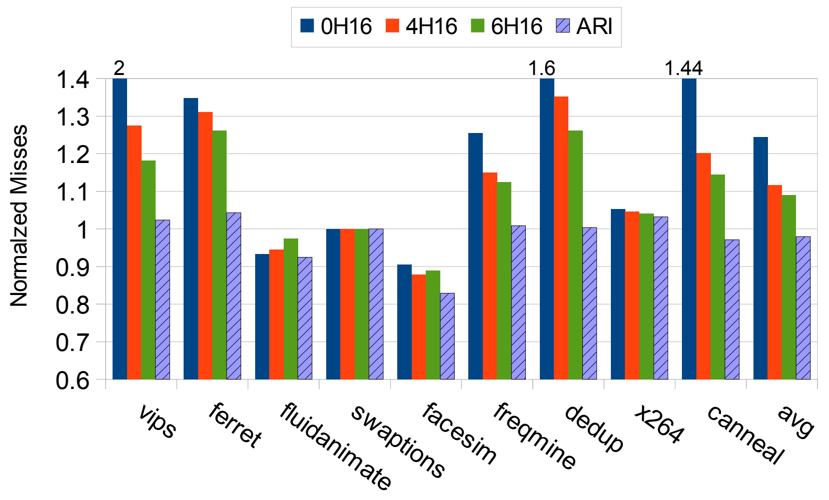


Fig. 9: Cache misses for the PARSEC applications, normalized to baseline LRU.

We also varied the size of the L3 cache, and found the writeback and miss rates improvement to be 27% and 0.9% on an average for an 8MB cache utilizing ARI, and 9% and 3.3% on an average for 32MB cache. The results for multiprocessor simulation are consistent with the single-core results (Figures 5, 6).

We expect that multi-program workloads, where distinct applications run on different cores, will behave similar to muti-threaded workloads. ARI generally adapts to the phase behavior observed in the shared last-level cache. This should hold true if the behavior is due to a single application, or the aggregate of many applications.

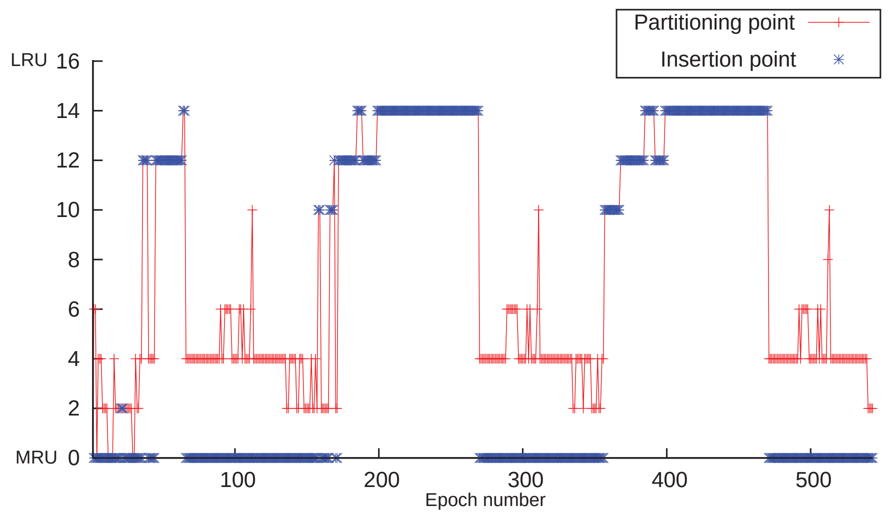


Fig. 10: Adaptivity graph for soplex application.

5. ANALYSIS

In this section we first discuss the dynamic behavior of the proposed scheme. Next, we discuss the effect of several parameters on the performance of the proposed scheme. Finally we discuss various means to reduce the hardware overhead of the technique.

5.1. Dynamic Behavior

Figure 10 shows how ARI adapts to program phases in soplex application; for each epoch, we plot the best stack distance distribution High-hit ways number (the n in our $n\mathrm{H}m$ notation) and the insertion policy used (the higher marks denote Low-hit insertion is used, and lower marks - conventional MRU insertion). As the figure shows, the best policy changes frequently over the execution of a program. ARI adapts the policy based on these observed program phases.

Figure 11 shows the writeback rate curves for mcf application (the X axis shows the period number, where we picked the data in intervals of 1 million LLC accesses for this graph; and Y axis shows the number of writebacks for the corresponding period). We observe that the program behavior can change fairly fast from one epoch to the next epoch. We see that ARI achieves lower writeback rate than the best static policy, 0H16, because it adapts insertion *and* replacement policy to the most beneficial stack distance distribution for the current phase. Our simulations show that using dynamic insertion in ARI yields an 8% writeback reduction on an average compared to the static MRU insertion.

5.2. Cache size and set-associativity

We varied the L3 cache size to determine the scalability of ARI. Figure 12 shows results for different cache sizes, in single-core configuration. The miss and writeback counts are normalized to the counts of the respective LRU-managed caches. It is clear that ARI scales well. We observed that (a) at 1MB cache size, misses are more frequent, data does not stay in the cache long enough to benefit from another write hitting in the cache before getting evicted; (b) at 8MB, it is more difficult to improve on LRU since due to the larger cache capacity, writebacks are becoming relatively infrequent;

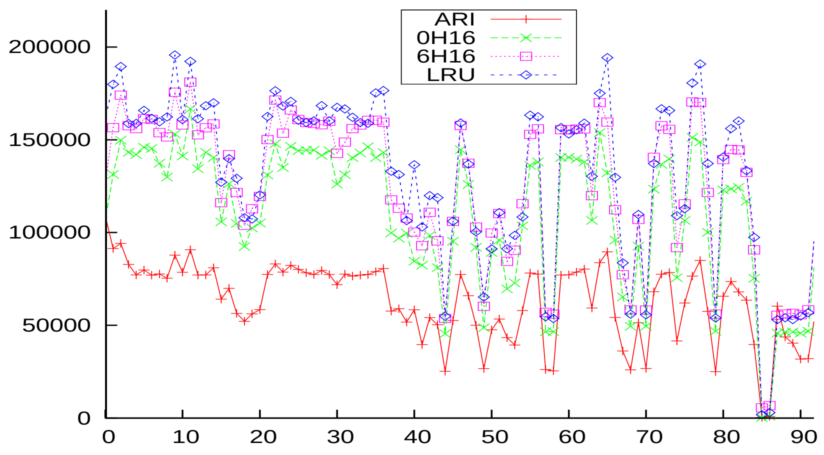


Fig. 11: Writeback curves for mcf, ARI vs. static policies.

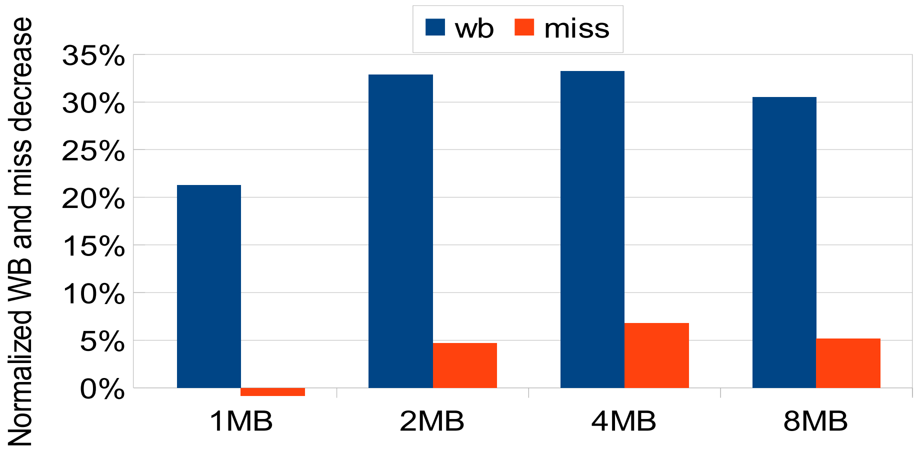


Fig. 12: Normalized Writebacks and Misses decrease under varied cache sizes, SPEC2006 (higher is better).

(c) ARI works the best with L3 caches 2-4MB in size, for the workloads considered in the article, yielding a decrease in writebacks of 33.3% on an average, and a decrease in misses of 6.8% on an average, as compared to baseline LRU.

We simulated 2MB LLCs with associativity from 4 up to 32 ways as shown in Figure 13. The number of policies per sample set is, respectively, 3, 5, 9, and 17. The larger number of ways provide more potential eviction candidates in Low-Hit ways according to the stack distance, thus more opportunities to gain from the sampling scheme. We noticed that the improvements in writeback reduction are around 3% when doubling the LLC associativity.

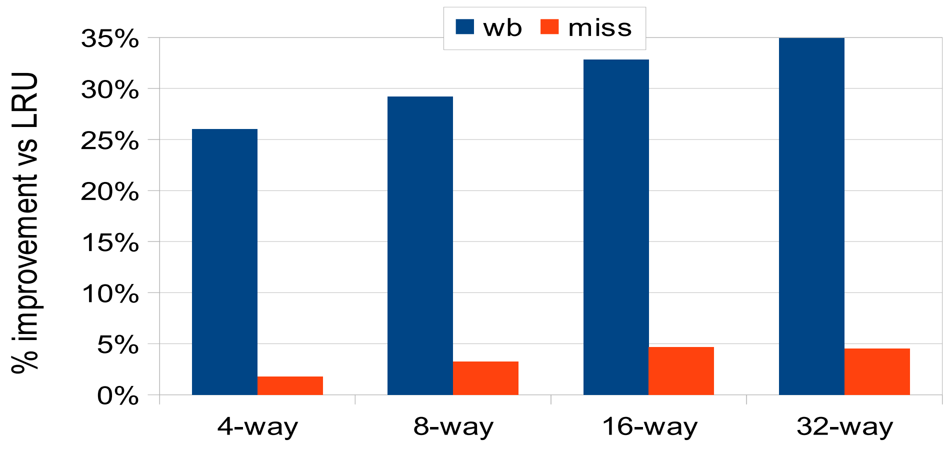


Fig. 13: Normalized Writebacks and Misses decrease under varied associativity (higher is better).

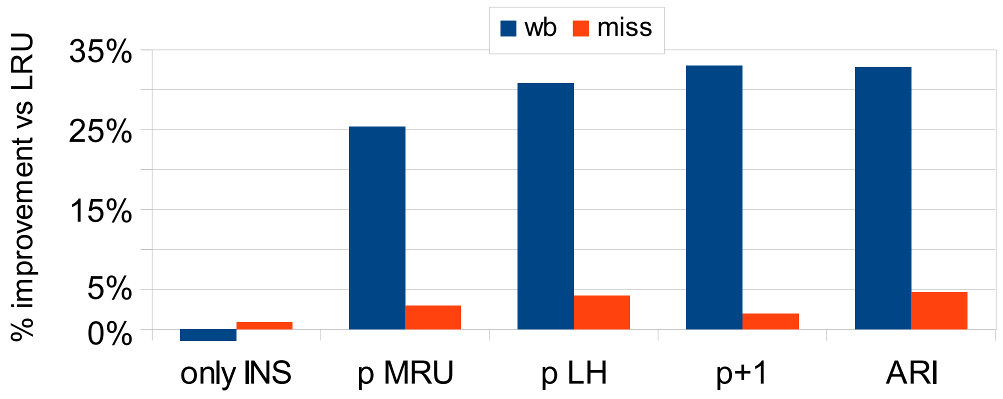


Fig. 14: Comparison of the schemes with various insertion policies.

5.3. Impact of insertion policies

Applications with constrained hardware budgets may halve the shadow tag hardware by adopting a simpler version of ARI, with relatively low impact on performance.

We performed three experiments with the following policies: 1) “Insertion-only” policy (no adaptive replacement) where sample sets are used to determine the best partition sizes, but then only the incoming cache blocks are inserted dynamically, while evicting only from LRU position; 2) ARI with fixed insertion, i.e. only MRU-, or only LH-insertion; 3) a '$p + 1$ scheme' which implements 9 sample policies with alternating insertion (i.e. 0H16-MRU, 2H16-LH, ..., 16H16-MRU) and one additional 'flipped' policy which assumes the same stack distance distribution as the best policy for a given epoch, but the insertion is inverted, i.e. if the best policy is 2H16-LH, then the corresponding flipped policy will be 2H16-MRU. The intent is to allow coverage of the entire space of 2*p policies using only $p + 1$ actual samples.

Insertion-only policy does not perform well, only improving misses by 1%, while increasing writebacks by 1.5% beyond LRU. This is because the sampling is done with adaptive ARI policies, which may have different best partition sizes than conventional LRU eviction policy.

We found that using ARI with LH-insertion yields an 8% writeback reduction compared to ARI with static MRU-insertion.

The $p + 1$ scheme reduces writebacks as well as ARI, although it sacrifices 2% miss reduction. These results indicate that the adaptive insertion provides a small but significant boost in ARI in terms of both miss rate and writebacks, though it comes at the cost of doubling the number of sample sets.

5.4. Minimizing hardware overhead

We examined a number of options to minimize the hardware overhead. We summarize the results here.

(1) We examined a simple hashing of the tag bits in the sampled sets, such that the top six bits of the tag is XOR'ed with the bottom six bits, generating a tag of 1/2 the original size (i.e. 6 bits instead of 12 bits). This technique impacted writebacks and misses by $< 1\%$ and is used for all results in Sections 4 and 5.
(2) Another way to reduce overhead is to utilize live sampled sets [Qureshi et al. 2007], each of which implements a static policy, instead of shadow tags as we propose. Simulations show this approach to increase writebacks by 5% and misses by 0.8% comparing to the shadow-tag approach.
(3) We varied the number of sampled sets used between 72 and 9, keeping the number of sample static policies at 9. We found that the difference is within 1.5% for writebacks and 2% for misses. Using 9 sampled sets instead of 72 yields a 8x decrease in the shadow tag storage.
(4) We varied the sampling epoch length between 10k and 40k cache references. Though the difference in performance was not significant, we found the maximum reduction in both miss rate and writebacks occurred for epochs of 25K references.
(5) We evaluated an interpolation approach where we only sample the data for p/2 policies each half-epoch, and then reconstruct the interpolated data for the full p policies. This reduces the sampling hardware by 50%. We found this approach to be inferior to the fixed LH insertion scheme.
(6) We varied the number of sample policies from p = 9, to 5, and to 3. We found that, compared to p = 9, the scheme with 5 policies is only 2% behind in writebacks and 1.2% behind in misses on an average, while the scheme with 3 policies is 6% and 3.2% behind, respectively, which we believe is reasonable considering the 3x reduction in the number of sample policies.

6. RELATED WORK

A number of studies have been dedicated to improving the performance of cache replacement policies [Jouppi 1993; Lee et al. 1999; Subramanian et al. 2006; Jaleel et al. 2010b]. Gao and Wilkerson [2010] dynamically select two variants of Segmented LRU algorithm with insertion bypassing. Michaud [2010] modified the DIP scheme for use with the CLOCK algorithm [Corbató 1968], and used 4 dueling policies as opposed to 2 in DIP. Khan and Jimenez [2010] extended the DIP scheme to multilevel dueling, decreasing the number of leader sets. But this requires several "rounds" of dueling and switching the policies in the leader sets. Ishii et al. proposed to vary the insertion positions of incoming lines in the LRU stack based on their observed reuse possibility [Ishii et al. 2010]. This scheme utilizes set dueling and has an additional overhead of 8kB beyond DIP, and requires communication circuitry from CPU core to the LLC. RRIP [Jaleel et al. 2010b] "predicts" the reuse interval. It uses set dueling to choose between two NRU-based policies and gains 4% speedup on an average. In contrast, our scheme uses sampling between $2p$ policies and has 4.9% average speedup. We compared ARI against one of the best-performing LLC schemes, the Deadblock-predictor [Khan et al. 2010], and showed that our scheme is superior in reducing the writebacks, which is very important in PCM-based systems. These recent works mostly disregard the effect of writebacks on the DRAM, and put focus on reducing cache misses. In contrast, our scheme aims at reducing writebacks while simultaneously decreasing the cache miss rate.

With the embedded designs placing greater pressure on the memory system hierarchy, this focus will have to change as off-chip bandwidth becomes a highly constrained commodity. Additionally, writes in the main memory can interfere with reads [Lee et al. 2010b]. Furthermore, as power consumption becomes an important issue in processor design, the extra power required by main memory writes makes reducing writebacks advantageous regardless of the main memory implementation technology.

Goodman discussed the impact of writebacks on memory system bandwidth [Goodman 1983]. Clean First LRU (CFLRU) replacement policy for page cache of SSDs [Park et al. 2006], has similar motivation with ours in reducing expensive writes. However, our focus is on appropriate last-level, on-die cache, block replacement policies for main memory. Further, if CFLRU were mapped to CPU LLC, it would roughly correspond to our *nH16* static policy; we showed that ARI outperforms all static policies in reducing misses and writebacks. Wang et al. proposed a Last-Write prediction LLC policy [Wang et al. 2012] where the dirty blocks predicted to not receive any more writes are speculatively written back to memory before they reach the LRU position. This scheme can be incorporated into ARI low-hit partition to give two benefits: 1) more intelligent way of breaking the ties in case the partition is full with dirty blocks; 2) more clean blocks in the low-hit partition allows for more room to store other low-hit dirty blocks.

The idea of set sampling has been fruitfully used in the literature [Jiang et al. 2011; Khan et al. 2010; Janapsatya et al. 2010]. Hybrid cache insertion uses set dueling [Qureshi et al. 2007] to dynamically choose between multiple insertion policies. The previous set sampling policies examine the LRU stack insertion points for new lines, while we attempt to determine the best region in the set for replacement *as well as* insertion.

DRAM-aware LLC management policies have been proposed [Lee et al. 2000; Lee et al. 2010b; Lee et al. 2010a; Stuecheli et al. 2010]. Eager writeback [Lee et al. 2000] writes dirty cache blocks to memory during idle cycles, to provide more clean blocks as eviction candidates, effectively shifting the writes in time. Virtual write-queue [Stuecheli et al. 2010] uses a fraction of the LLC as a write queue for the memory, and the writebacks from the cache are governed by DRAM-controller. Lee et al. [2010b; 2010a] exploit row-buffer locality to guide the eviction decisions and minimize the write-caused interference. The memory-aware schemes can be easily integrated with ARI in the following way. The high-hit partition can be left managed by LRU, while memory-aware policies can be utilized in the low-hit partition to further decrease the bandwidth consumption and write interference in DRAM systems. While these techniques can be adopted in DRAM-based main memory, they are detrimental to the lifetime of PCM-based systems as they increase the number of writebacks by an average of 5-10% beyond the conventional LRU scheme [Stuecheli et al. 2010]. Write interference in PCM memories has been shown to be a minor issue as the writes can be paused to give way to reads [Qureshi et al. 2010].

PCM is receiving significant attention for use within memory hierarchy. Hybrid DRAM+PCM memory architectures have been investigated [Wu et al. 2009; Dhiman et al. 2009]. Wear leveling algorithms have been proposed to distribute writes uniformly [Qureshi et al. 2009a; Seong et al. 2010]. Write reduction techniques [Qureshi et al. 2010] and techniques for improving PCM lifetime [Lee et al. 2010; Lee et al. 2009; Ferreira et al. 2010] have been proposed for hardware implementation. Ramos et al. [2011] and Yoon et al. [2012] have presented smart page placement techniques for hybrid memory, based on "popularity" and row buffer hit ratios, respectively. Loh and Hill [2012], Meza et al. [2012], and Qureshi and Loh [2012] explored hardware support for large-size, fine-granularity DRAM caches, where conventional architecture is impractical due to huge SRAM tag storage overhead. By storing tags in the DRAM itself, they decrease the cache access latency, thus boosting the performance. However, large DRAM caches may be inefficient in mobile applications due to high static power consumption. ARI implemented in L3 cache helps reduce the number of accesses to such DRAM cache, which may help reduce the energy consumption and boost the performance of hybrid architectures. These works on DRAM+PCM hybrid architectures are largely orthogonal and possibly complementary to the work presented here, as we focus upon reducing writebacks from the lower levels in the memory system. Furthermore, by adapting insertion policy as well as replacement policy, ARI further reduces writebacks while reducing misses.

7. CONCLUSION

In this article we presented ARI, a technique for dynamic cache management capable of reducing the memory system bandwidth, by optimizing both misses and writebacks at the same time. We have shown ARI to perform well under various workloads, such as single-threaded and multithreaded applications in a CMP. ARI provides 33% LLC writeback reduction and 4.7% miss rate reduction on an average, yielding 8.9% memory bandwidth improvement, and 4.9% IPC speedup. We find that a PCM-based system with an LLC utilizing our scheme uses 8.9% less energy, and enjoys a 49% lifetime improvement on an average, as compared to conventional LRU.

In the future we plan to explore further adaptation schemes and optimize the stack distance estimation. Also, it seems possible to employ different metrics in selecting a cache replacement policy through sampling.

ACKNOWLEDGMENTS

The authors would like to thank Onur Mutlu and the anonymous reviewers for their insightful comments which helped to improve the quality of this manuscript.

REFERENCES

Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. DOI: http://dx.doi.org/10.1145/2024716.2024718

Fernando J. Corbató. 1968. *A paging experiment with the multics system*. Technical Report. DTIC Document.

Gaurav Dhiman, Raid Ayoub, and Tajana Rosing. 2009. PDRAM: A Hybrid PRAM and DRAM Main Memory System. In *Proceedings of the 46th Annual Design Automation Conference (DAC '09)*. ACM, New York, NY, USA, 664–469. DOI: http://dx.doi.org/10.1145/1629911.1630086

Alexandre P. Ferreira, Miao Zhou, Santiago Bock, Bruce Childers, Rami Melhem, and Daniel Mossé. 2010. Increasing PCM Main Memory Lifetime. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '10)*. European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 914–919. http://dl.acm.org/citation.cfm?id=1870926.1871147

Hongliang Gao and Chris Wilkerson. 2010. A Dueling Segmented LRU Replacement Algorithm with Adaptive Bypassing. In *JWAC 2010-1st JILP Worshop on Computer Architecture Competitions: Cache Replacement Championship*.

Jayesh Gaur, Mainak Chaudhuri, and Sreenivas Subramoney. 2011. Bypass and Insertion Algorithms for Exclusive Last-level Caches. In *Proceedings of the 38th Annual Internation-*

al Symposium on Computer Architecture (ISCA '11). ACM, New York, NY, USA, 81–92. DOI:http://dx.doi.org/10.1145/2000064.2000075

James R. Goodman. 1983. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th annual international symposium on Computer architecture (ISCA '83)*. ACM, New York, NY, USA, 124–131. DOI:http://dx.doi.org/10.1145/800046.801647

International Technology Roadmap for Semiconductors (ITRS) Working Group. 2009. International Technology Roadmap for Semiconductors (ITRS), 2009 Edition. (2009). http://www.itrs.net/Links/2009ITRS/Home2009.htm

Yasuo Ishii, Mary Inaba, Kei Hiraki, and others. 2010. Cache replacement policy using map-based adaptive insertion. In *JWAC 2010-1st JILP Worskhop on Computer Architecture Competitions: Cache Replacement Championship*.

Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C. Steely Jr., and Joel Emer. 2010a. Achieving Non-Inclusive Cache Performance with Inclusive Caches: Temporal Locality Aware (TLA) Cache Management Policies. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43)*. IEEE Computer Society, Washington, DC, USA, 151–162. DOI:http://dx.doi.org/10.1109/MICRO.2010.52

Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. 2010b. High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP). In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA, 60–71. DOI:http://dx.doi.org/10.1145/1815961.1815971

Andhi Janapsatya, Aleksandar Ignjatović, Jorgen Peddersen, and Sri Parameswaran. 2010. Dueling CLOCK: Adaptive Cache Replacement Policy Based on the CLOCK Algorithm. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '10)*. European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 920–925. http://dl.acm.org/citation.cfm?id=1870926.1871148

Xiaowei Jiang, Asit Mishra, Li Zhao, Ravishankar Iyer, Zhen Fang, Sadagopan Srinivasan, Srihari Makineni, Paul Brett, and Chita R. Das. 2011. ACCESS: Smart scheduling for asymmetric cache CMPs. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*. 527–538. DOI:http://dx.doi.org/10.1109/HPCA.2011.5749757

Norman P. Jouppi. 1993. Cache Write Policies and Performance. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA '93)*. ACM, New York, NY, USA, 191–201. DOI:http://dx.doi.org/10.1145/165123.165154

Samira Khan and Daniel A. Jiménez. 2010. Insertion policy selection using Decision Tree Analysis. In *Computer Design (ICCD), 2010 IEEE International Conference on*. 106–111. DOI:http://dx.doi.org/10.1109/ICCD.2010.5647608

Samira Khan, Yingying Tian, and Daniel A. Jiménez. 2010. Sampling Dead Block Prediction for Last-Level Caches. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*. 175–186. DOI:http://dx.doi.org/10.1109/MICRO.2010.24

Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting Phase Change Memory As a Scalable Dram Alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM, New York, NY, USA, 2–13. DOI:http://dx.doi.org/10.1145/1555754.1555758

Benjamin C. Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. 2010. Phase-Change Technology and the Future of Main Memory. *Micro, IEEE* 30, 1 (2010), 143–143. DOI:http://dx.doi.org/10.1109/MM.2010.24

Chang Joo Lee, Onur Mutlu, Eiman Ebrahimi, Veynu Narasiman, and Yale N Patt. 2010a. *DRAM-Aware Last-Level Cache Replacement*. Technical Report TR-HPS-2010-007. High Performance Systems Group, Department of Electrical and Computer Engineering, The University of Texas at Austin.

Chang Joo Lee, Onur Mutlu, Veynu Narasiman, Eiman Ebrahimi, and Yale N Patt. 2010b. *DRAM-Aware Last-Level Cache Writeback: Reducing Write-Caused Interference in Memory Systems*. Technical Report TR-HPS-2010-002. High Performance Systems Group, Department of Electrical and Computer Engineering, The University of Texas at Austin.

Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. 1999. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems (SIGMETRICS '99)*. ACM, New York, NY, USA, 134–143. DOI:http://dx.doi.org/10.1145/301453.301487

Hsien-Hsin S. Lee, Gary S. Tyson, and Matthew K. Farrens. 2000. Eager writeback - a technique for improving bandwidth utilization. In *Proceedings of the 33rd annual ACM/IEEE in-*

ternational symposium on Microarchitecture (MICRO 33). ACM, New York, NY, USA, 11–21. DOI: http://dx.doi.org/10.1145/360128.360132

Gabriel H. Loh and Mark D. Hill. 2012. Supporting Very Large DRAM Caches with Compound-Access Scheduling and MissMap. *Micro, IEEE* 32, 3 (2012), 70–78. DOI: http://dx.doi.org/10.1109/MM.2012.25

Justin Meza, Jichuan Chang, HanBin Yoon, Onur Mutlu, and Parthasarathy Ranganathan. 2012. Enabling Efficient and Scalable Hybrid Memories Using Fine-Granularity DRAM Cache Management. *IEEE Comput. Archit. Lett.* 11, 2 (July 2012), 61–64. DOI: http://dx.doi.org/10.1109/L-CA.2012.2

Pierre Michaud. 2010. The 3P and 4P cache replacement policies. In *JWAC 2010-1st JILP Worshop on Computer Architecture Competitions: Cache Replacement Championship*.

Daniel Molka, Daniel Hackenberg, Robert Schone, and Matthias S. Muller. 2009. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques (PACT '09)*. IEEE Computer Society, Washington, DC, USA, 261–270. DOI: http://dx.doi.org/10.1109/PACT.2009.22

Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jin-soo Kim, and Joonwon Lee. 2006. CFLRU: A Replacement Algorithm for Flash Memory. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '06)*. ACM, New York, NY, USA, 234–241. DOI: http://dx.doi.org/10.1145/1176760.1176789

Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. 2003. Using SimPoint for accurate and efficient simulation. (2003), 318–319. DOI: http://dx.doi.org/10.1145/781027.781076

Moinuddin K. Qureshi, Michele M. Franceschini, and Luis A. Lastras-Montano. 2010. Improving read performance of Phase Change Memories via Write Cancellation and Write Pausing. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. 1–11. DOI: http://dx.doi.org/10.1109/HPCA.2010.5416645

Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. 2007. Adaptive Insertion Policies for High Performance Caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*. ACM, New York, NY, USA, 381–391. DOI: http://dx.doi.org/10.1145/1250662.1250709

Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. 2009a. Enhancing Lifetime and Security of PCM-based Main Memory with Start-gap Wear Leveling. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*. ACM, New York, NY, USA, 14–23. DOI: http://dx.doi.org/10.1145/1669112.1669117

Moinuddin K. Qureshi and Gabriel H. Loh. 2012. Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '12)*. IEEE Computer Society, Washington, DC, USA, 235–246. DOI: http://dx.doi.org/10.1109/MICRO.2012.30

Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009b. Scalable High Performance Main Memory System Using Phase-change Memory Technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM, New York, NY, USA, 24–33. DOI: http://dx.doi.org/10.1145/1555754.1555760

Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. 2011. Page Placement in Hybrid Memory Systems. In *Proceedings of the International Conference on Supercomputing (ICS '11)*. ACM, New York, NY, USA, 85–95. DOI: http://dx.doi.org/10.1145/1995896.1995911

Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Comput. Archit. Lett.* 10, 1 (Jan. 2011), 16–19. DOI: http://dx.doi.org/10.1109/L-CA.2011.4

Nak Hee Seong, Dong Hyuk Woo, and Hsien-Hsin S. Lee. 2010. Security Refresh: Prevent Malicious Wear-out and Increase Durability for Phase-change Memory with Dynamically Randomized Address Mapping. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA, 383–394. DOI: http://dx.doi.org/10.1145/1815961.1816014

Jeffrey Stuecheli, Dimitris Kaseridis, David Daly, Hillery C. Hunter, and Lizy K. John. 2010. The Virtual Write Queue: Coordinating DRAM and Last-level Cache Policies. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA, 72–82. DOI: http://dx.doi.org/10.1145/1815961.1815972

Ranjith Subramanian, Yannis Smaragdakis, and Gabriel H. Loh. 2006. Adaptive Caches: Effective Shaping of Cache Behavior to Workloads. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*. IEEE Computer Society, Washington, DC, USA, 385–396. DOI: http://dx.doi.org/10.1109/MICRO.2006.7

Zhe Wang, Samira M. Khan, and Daniel A. Jiménez. 2012. Improving writeback efficiency with decoupled last-write prediction. In *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*. 309–320. DOI:http://dx.doi.org/10.1109/ISCA.2012.6237027

Xiaoxia Wu, Jian Li, Lixin Zhang, Evan Speight, Ram Rajamony, and Yuan Xie. 2009. Hybrid Cache Architecture with Disparate Memory Technologies. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM, New York, NY, USA, 34–45. DOI:http://dx.doi.org/10.1145/1555754.1555761

HanBin Yoon, Justin Meza, Rachata Ausavarungnirun, Rachael A. Harding, and Onur Mutlu. 2012. Row buffer locality aware caching policies for hybrid memories. In *Computer Design (ICCD), 2012 IEEE 30th International Conference on*. 337–344. DOI:http://dx.doi.org/10.1109/ICCD.2012.6378661

Ying Zheng, B. T. Davis, and M. Jordan. 2004. Performance Evaluation of Exclusive Cache Hierarchies. In *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '04)*. IEEE Computer Society, Washington, DC, USA, 89–96. http://dl.acm.org/citation.cfm?id=1153925.1154591

Received February 2013; revised September 2013; accepted November 2013

Online Appendix to: ARI: Adaptive LLC-Memory Traffic Management

VIACHESLAV V. FEDOROV, Texas A&M University
SHENG QIU, Texas A&M University
A. L. NARASIMHA REDDY, Texas A&M University
PAUL V. GRATZ, Texas A&M University

© 2013 ACM 1544-3566/2013/01-ART0 $15.00
DOI: http://dx.doi.org/10.1145/0000000.0000000