Appears in Proceedings of the 39th Annual ACM/IEEE International Symposium on Computer Architecture (ISCA-2012), June, 2012

Improving Writeback Efficiency with Decoupled Last-Write Prediction

Zhe Wang   Samira M. Khan   Daniel A. Jiménez
The University of Texas at San Antonio
{zhew,skhan,dj}@cs.utsa.edu

Abstract

In modern DDRx memory systems, memory write requests compete with read requests for available memory resources, significantly increasing the average read request service time. Caches are used to mitigate long memory read latency that limits system performance. Dirty blocks in the last-level cache (LLC) that will not be written again before they are evicted will eventually be written back to memory. We refer to these blocks as last-write blocks. In this paper, we propose an LLC writeback technique that improves DRAM efficiency by scheduling predicted last-write blocks early. We propose a low overhead last-write predictor for the LLC. The predicted last-write blocks are made available to the memory controller for scheduling. This technique effectively re-distributes the memory requests and expands writes scheduling opportunities, allowing writes to be serviced efficiently by DRAM. The technique is flexible enough to be applied to any LLC replacement policy. Our evaluation with multi-programmed workloads shows that the technique significantly improves performance by 6.5%-11.4% *on average over the traditional writeback technique in an eight-core processor with various DRAM configurations running memory intensive benchmarks.*

1 Introduction

Memory access latency is a major performance bottleneck. A last-level cache miss can stall the pipeline and require hundreds of cycles of delay. Memory write requests compete with read requests for the available memory resources, increasing the average service time of read requests. This write-induced interference [12] has a significant impact on system performance.

There are two aspects to reducing write-induced interference. First, we must consider when to schedule the write requests [27]. System performance is sensitive to memory read latency, so write requests should be scheduled to have minimal interference with read requests. Second, we must consider how to schedule write requests. Write requests should be scheduled so that they can be serviced by DRAM efficiently. Scheduling can benefit from a large *write scheduling space* that can be used to hold the write scheduling candidates. A large write scheduling space can improve writeback efficiency by increasing the possibility of scheduling row-buffer hits and exploiting bank-level parallelism, i.e., write requests may be serviced in parallel in different DRAM banks.

The traditional way to handle writeback requests is to write a dirty block into the write buffer when it is evicted from the LLC. Write requests in the write buffer are scheduled for service based on the buffer management policy. Since the write buffer only has a small number of entries, the ability to schedule write requests with good locality is limited. Thus, the scheduling decision made is far from optimal. Increasing the size of the write buffer can expand the write scheduling space of the memory controller, but a large write buffer is complex and power inefficient. Moreover, exposing dirty blocks to the memory controller when they are evicted causes high contention between bursty reads and writes by clustering memory traffic, thus reducing memory bandwidth utilization and degrading performance [13].

Previous work [13, 27, 12] proposes to write back dirty cache blocks near the least-recently-used (LRU) position early. Though these proposals reduce write-induced interference by either balancing the memory bandwidth or expanding the memory scheduling space, they depend on the recency levels of LLC replacement policies such as LRU. However, such policies are prohibitively expensive in the LLC, and these writeback techniques cannot work with less costly replacement policies with no distinct recency levels, such as not recently used (NRU) and random replacement policy.

This paper proposes a decoupled LLC writeback technique. The technique reduces write-induced interference by scheduling predicted *last-write blocks* early. A last-write block is a dirty block in the LLC that will not be written again before it is evicted. Thus, last-write blocks in the LLC are available for scheduling without incurring extra memory write requests. We propose a low overhead *last-write predictor* (LWP) based on sampling. A last-write buffer is used to track the predicted last-write blocks until they are sent to DRAM. Experimental results show our technique significantly reduces write-induced interference and improves DRAM efficiency. The technique is independent of LLC replacement policy so it is flexible enough to apply to any replacement policy.

This paper makes the following contributions:

- We propose a low overhead last-write predictor based on sampling. This predictor uses a low-overhead LLC write simulator to simulate and make predictions based on the write behavior of the LLC. We demonstrate that the proposed last-write predictor can accurately predict last-write blocks.

- We propose a decoupled LLC writeback technique that makes last-write blocks in the LLC available to the memory controller for scheduling. The technique effectively expands the write scheduling space and balances memory bandwidth by re-distributing memory write requests, thus reducing write-induced interference. The technique is completely decoupled from the LLC replacement policy.

- We present an evaluation of our technique with multi-programmed SPEC CPU2006 workloads simulated with the MARSSx86 [19] simulator together with DRAMSim2 [23]. Our evaluation simulating an eight-core processor shows that the technique improves performance by 6.5%-11.4% on average over the traditional writeback technique with various DRAM configurations and LLC replacement policies. Our technique also significantly improves system performance in the presence of prefetching.

2 Background and Related Work

2.1 DRAM Memory Systems

The DDRx based memory system [2, 4] consists of one or more dual in-line memory modules (DIMMs) composed of multiple chips. Each chip is organized as multiple banks that can be operated in parallel. A memory rank is made up of a set of chips where chips in the same rank can be accessed simultaneously. In a DDRx memory module, each rank has a 64-bit data bus. Chips within a rank work in unison to return 64 bits per cycle. The memory channel is made up of one or multiple memory ranks. Ranks in the same channel share the same data bus. Modern multicore processors may have multiple channels.

A memory access includes both row access and column access [4]. An entire row of bits that contains the required data is brought into the row buffer during row access, then a column of this row buffer is selected according to the column address. Memory access requests may be row-buffer hit requests, row-buffer closed requests, or row-buffer conflict requests. A row-buffer hit request goes to a currently open row. Data can be accessed without activating the row buffer again. A row-buffer closed request goes to a row when there is no open row in the row buffer. The required row must be activated before the data in the row-buffer can be accessed. A row-buffer conflict request goes to a row other than the currently open row. Data in the currently open row must be written back first, then the required row must be activated before the data can be accessed. Thus, the access latency for row-buffer conflict/closed requests is significantly higher than for row-buffer hit requests.

2.2 Related Work

2.2.1 Memory Access Scheduling

Memory access scheduling [22] reorders memory references to improve memory performance. Much previous work [24, 28, 17, 1, 16, 9, 18] focuses on improving memory efficiency by intelligently scheduling memory requests. Shao *et al.* [24] propose a burst scheduling algorithm that schedules requests that hit in the same row buffer into a burst to increase row buffer hit rates and bus utilization. Sudan *et al.* [28] propose a page migration algorithm that collocates frequently accessed data in the same row buffer to increase row buffer hit rates in a multi-core system. Mutlu *et al.* [17] propose a parallelism-aware batch scheduling technique for multi-core systems. Their technique first organizes memory requests into batches to ensure the fairness of service, then within each batch, requests are scheduled to maximize parallelism while at the same time minimizing the number of idle cores by using a shortest-job-first scheduling technique. Ipek *et al.* [9] use a reinforcement-learning approach to learn the optimal memory scheduling policy according to past behavior.

2.2.2 Last-Level Cache Writeback

Much previous work [28, 17, 1, 16, 18] does not take into account the write-induced interference problem. Eager writeback [13] is the first proposal that increases the visibility of the write buffer by using the LLC to reduce write-induced interference. Eager writeback writes back dirty cache blocks in the LRU position of the LLC sets whenever the bus is idle instead of waiting for the block to be evicted to reduce the memory traffic. However, the scheduling space in eager writeback is still limited to the write buffer.

Stuecheli *et al.* [27] propose a virtual write queue (VWQ) technique. Their technique takes a fraction of the LRU positions in the LLC as the virtual write queue (also requiring LRU). Dirty cache blocks in the virtual write queue that target the same row buffer when mapping to the memory resource will be written back in a batch, therefore reducing write-induced interference. In their paper, they also propose the concept of scheduled writeback. Scheduled writeback writes back the dirty cache blocks that exploit locality in the DRAM structure to improve the memory performance. Chang *et al.* [12] propose a similar technique that writes back qualified dirty cache blocks in the LLC to improve the memory efficiency.

2.2.3 Dead Block Prediction

Lai *et al.* [11] proposes last touch predictor that predicts the last touch cache blocks for core caches. The last touch predictor uses program counter (PC) traces to detect the last touch and invalidate the shared cache blocks to reduce cache coherence overhead. Several dead block predictors are proposed in previous work [3, 7, 14, 10]. The trace-based dead block predictor [11] can detect when a cache block is accessed for the last time based on the a given sequence of memory-access PCs. This predictor is used to prefetch data into dead blocks in the L1 data cache. Hu *et al.* [7] propose a time based dead block predictor that learns the number of cycles a block is live and predicts it dead if it is not accessed for twice that number of cycles. This predictor is used to prefetch into the L1 cache and filter a victim cache. Recent work proposes [10] sampling dead block predictor for LLC that predict the dead blocks in the LLC and replace them for useful cache blocks.

3 Motivation

Previous proposals [12, 27, 20] point out that write-induced interference can degrade system performance significantly. Write-induced interference is a more severe problem in CMP systems since performance of an application is affected not only by its own write requests, but also write requests from other applications.

Figure 1 shows the simulation result for traditional writeback policy with various buffer sizes and perfect writeback. The system configuration for this experiment is a 4.8GHZ eight-core processor and a DDR3-1600 memory. The DRAM system has two channels/memory controllers, each memory controller has a per-channel write buffer. Each memory channel is made up of one rank. We run six randomly mixed multi-programmed SPEC CPU 2006 workloads; each benchmark runs simultaneously with the others. The result of the 16 memory intensive benchmarks is shown in the graph. The traditional writeback policy (i.e. writes go to the write buffer on LLC eviction) with various write buffer sizes and the perfect writeback are evaluated. The write buffer management policy we used for the traditional writeback policy is for writes in the write buffer to be scheduled for service whenever either of the following two conditions is satisfied: 1) the corresponding rank is idle and the number of writes in write buffer reaches a threshold, or 2) the write buffer is full. Perfect writeback assumes that write requests do not cause any interference with read requests. In Figure 1, the performance of the traditional writeback policy with a practical 32-entry write buffer is taken as the baseline. The graph shows that system performance improves 25.4% without write-induced interference. Thus, there is significant headroom for writeback optimization to improve system performance.

Figure 1 also shows that system performance improves as the size of the write buffer increases. A large write buffer can hold more write requests so the possibility of row-buffer hits and bank-level parallelism increases. For a practical 32-entry buffer size, the average row-buffer hit rate for writes is only 34.2% because, when dirty cache blocks are evicted from the LLC, their spatial locality has been filtered by core caches and the LLC. Additionally, in a CMP system, row-buffer utilization can be reduced by the distribution of requests from different applications. Thus, a small buffer size has a limited ability for preserving row-buffer hit requests. Increasing the size of the write buffer also increases the row-buffer hit rate since more scheduling candidates in the write buffer offers more opportunities for row-buffer hit requests. For a 512-entry write buffer, the average row-buffer hit rate for writes is increased to 54.5%.

However, a large write buffer is complex and power-hungry, as each memory read request must associatively search the write buffer for a matching address. Thus, increasing the number of write buffer entries also significantly increases the on-chip power consumption. Therefore, building a large write buffer is not a practical way to improve memory efficiency. Additionally, exposing dirty cache blocks to the memory controller when they are evicted from the LLC can cause high bus contention between writes and reads. LLC misses tend to occur in bursts, so writes compete for memory bandwidth with bursty reads, reducing the memory bandwidth utilization by clustering memory traffic.

Previous work [27, 13] proposes to write back dirty cache blocks near the LRU position in the LLC earlier to reduce write-induced interference. These techniques depend on a policy that divides the LLC blocks of a set in levels of distinct recency (LRU and Pseudo LRU) to detect last-write cache block. They can not be adapted to less costly replacement policies like NRU and random.

We propose a low overhead LLC writeback technique. Instead of detecting last-write cache blocks using LRU recency information, this technique uses a last-write predictor based on sampling to predict the last-write cache blocks in LLC. Making the last-write cache blocks available to the memory controller early effectively re-distributes memory

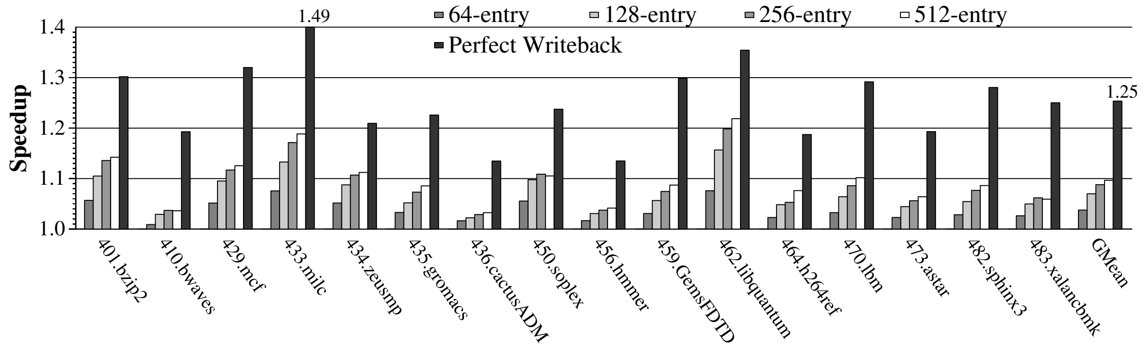


Figure 1. Speedup for traditional writeback with various write buffer sizes and perfect writeback

write requests and increases memory scheduling resources. Thus, DRAM efficiency is improved and the write-imposed penalty to following reads is reduced. This technique can work with any replacement policy without significant overhead.

4 Last-Write Predictor Guided Last-Level Cache Writeback

We propose a last-write predictor guided (LWPG) LLC writeback policy. Figure 2 shows the structure of our technique. A last-write predictor is proposed to predict last-write blocks when they access the LLC. A last-write buffer is used to track predicted last-write blocks. Write requests in the last-write buffer as well as the write buffer are available to memory controller for scheduling. The LWPG writeback policy has the following advantages: 1) redistributing the memory requests and balancing the memory bandwidth, 2) expanding the scheduling space of memory controller, maintaining row-buffer hit and bank-level parallelism locality, and 3) completely decoupling from cache replacement policy allowing it to be applied to any LLC replacement policy.

4.1 Last-Write Predictor

The last-write predictor is used to predict last-write blocks in the LLC. It is composed of a lightweight LLC write simulator and a prediction table. Once a dirty block is evicted from the core cache and accesses the LLC, the last-write predictor consults the prediction table to make a prediction. The instruction PC related to the dirty block is hashed to index the prediction table to get the prediction result. An LLC write simulator is used to update the prediction table according to the simulated write behavior of the LLC.

The last-write predictor is a PC-based predictor. It is based on the observation that if an instruction PC leads to the last write access to one block, then there is a high probability that the next time this instruction is reached it will also lead to a last-write block. For a writeback cache, once a dirty block is evicted from the core cache, it has no PC information with it. Thus, a PC field is associated with each core block. Once a write accesses the core cache, the PC related to this write will be stored with the block.

4.1.1 Prediction Table

The prediction table uses skewed organization [10, 15] to reduce the impact of conflicts in the table. It consists of three tables, each indexed by a different hash of 16-bits partial PC. Each entry in the table has a two-bit saturating counter. Once a dirty block is evicted from the core cache and accesses the LLC, the LWP predicts whether or not this dirty block is a last-write block. The prediction decision is based on the sum of the counter values for all three tables that indexed by different hashes of the PC related to this dirty block: if the sum is greater than a threshold, then it is a last-write block. The prediction table is updated by the LLC write simulator.

4.1.2 LLC Write Simulator

The LLC write simulator simulates the write behavior of the LLC and updates the prediction table. To reduce overhead, only a few sets of the LLC are represented. LLC sets are sampled; there is one simulated set for every 16 cache sets. Only partial tags are represented since simulator correctness is not required; in practice, we find 16 bits of tag leads to $>99\%$ accuracy with respect to full tags. Of course, no data are represented in the simulated sets. The LLC write simulator only simulates the write behavior of the LLC, i.e. missing reads from memory are not placed in the simulator. The write accesses of the LLC account for about $1/3$ of total number of accesses on average in the memory intensive SPEC CPU 2006 benchmarks. Thus, the write simulator can use a smaller associativity compared with the LLC. The

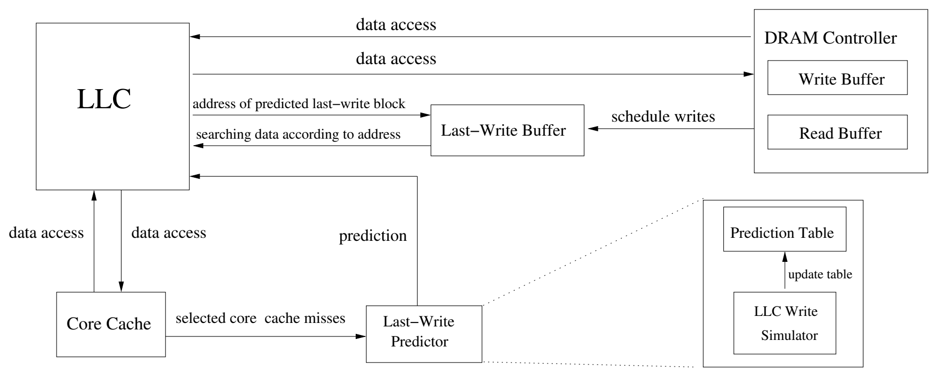


Figure 2. System structure

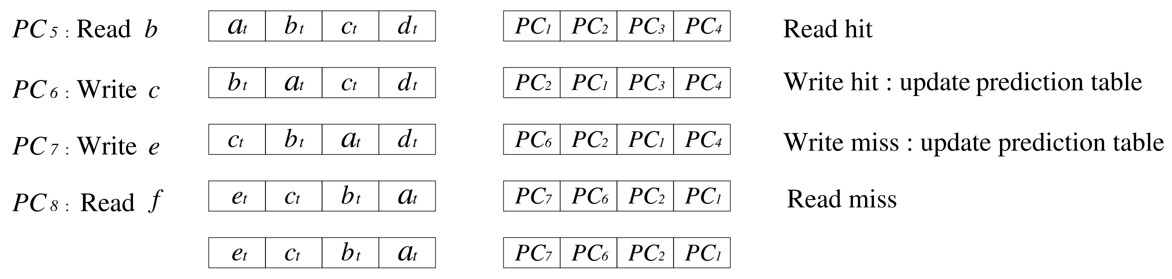

PC_5 : Read b
a_t b_t c_t d_t
PC_1 PC_2 PC_3 PC_4
Read hit
PC_6 : Write c
b_t a_t c_t d_t
PC_2 PC_1 PC_3 PC_4
Write hit : update prediction table
PC_7 : Write e
c_t b_t a_t d_t
PC_6 PC_2 PC_1 PC_4
Write miss : update prediction table
PC_8 : Read f
e_t c_t b_t a_t
PC_7 PC_6 PC_2 PC_1
Read miss
e_t c_t b_t a_t
PC_7 PC_6 PC_2 PC_1

Figure 3. Behavior of the LLC write simulator

associativity of the LLC simulator is 6 while the associativity of the LLC is 16. Each entry in the simulator set has a partial tag field, a partial write PC field, a valid bit and an LRU recency field. When a write accesses a sampled LLC set, it also accesses the simulator simultaneously. The corresponding sampled set is searched for an entry with a matching tag; if there is a miss in the simulator, an entry is allocated using an LRU victim entry. LRU is used in the simulator, but since the associativity and number of sampled sets are low, the implementation of LRU is far more feasible than in the LLC [10]. The simulator also updates the prediction table. When a read accesses the simulator, if it is a hit, the LRU recency will be updated. If it is a miss, the simulator will do nothing. Read access to the simulator updates the recency information for synchronizing the behavior of the simulator with the LLC, while the write access also needs to update the predictor.

Figure 3 illustrates the set behavior of the write simulator. Assuming a four-entry set, the box on the left side shows the LRU stack of the partial tag field. The box on the right side shows the partial write PC corresponding to the same entry with the partial tag on the left side. The PC for write access on the left in Figure 3 is the partial PC related to the evicted dirty block from the core cache.

At beginning, partial tags a_t, b_t, c_t, d_t of blocks a, b, c, d and their related PCs are reside in the set entries. First, request “read b” accesses the simulator, it is a read hit, so it updates the LRU recency of block b to the MRU position. Since it is a read access, the prediction table is not updated. Then, request “write c” accesses the simulator. It is a write hit meaning that PC_3 leads to a dirty block that could be rewritten again before it is evicted. Thus, we update the entry in prediction table that indexed by PC_3 using ’not last-write’, and update the LRU recency of block c to MRU position. Then request “write e” accesses the set. It is a miss, so we replace d with e since PC_4 leads to a last-write block d that did not access again before it is evicted. Thus, we update the entry in prediction table that indexed by PC_4 using ’last-write’. Finally, request “read f” accesses the set. It is a read miss, so the simulator does nothing.

The write simulator itself uses LRU replacement policy, but it can also accurately simulate the last-write behavior for LLC with other replacement policies. Write accesses to the write simulator and LLC are the same, thus they have same behavior. Though the replacement policy in LLC and write simulator may differ, a dirty block in the write simulator with LRU replacement policy that will not be accessed again before it is evicted also has a high probability that it will not be accessed again in LLC. Thus, the last-write predictor is independent of the LLC replacement policy.

4.2 Writeback Mechanism

4.2.1 Last-Write Buffer

In our technique, two buffers are used to hold write requests: the write buffer and the last-write buffer. The evicted dirty blocks are placed in the write buffer. The last-write buffer is used to track the predicted last-write blocks in the LLC. When the predictor predicts a last-write block, the physical address of the predicted last-write block will be placed into the last-write buffer. The write requests in the write buffer and the last-write buffer are available for scheduling. Since each entry in the last-write buffer only contains a 64-bit physical address, the data for the write requests are still in the LLC. Thus, memory read requests do not need to search the last-write buffer for address matching. This allows the last-write buffer to have many more entries than the write buffer. In our experiment, we use a 256-entry/channel (256-entry/c) per-rank last-write buffer, i.e. the last-write buffer is organized by rank and the total number of write buffer entries for a channel is 256 entries.

4.2.2 Priority Mechanism

A write buffer with infinite size would be able to always prioritize reads over writes, thus eliminating all write-imposed penalty on the following reads. Given a finite write buffer, it is better to prioritize writes over reads whenever writes will cause less interference to subsequent reads. In our technique, the service of write requests prioritizes read requests whenever either of the following conditions is satisfied: 1) the rank is idle and the write buffer has more active entries than a threshold m, or the last-write buffer has more active entries than a threshold n, 2) the write buffer or the last-write buffer is full. Condition 1) is to fill rank idle cycles with writes, reducing the contention between reads and writes. In condition 2), to ensure the progress of the application, scheduled writes in write buffer must be sent to DRAM for service when the write buffer is full to avoid pipeline stalls. Once the last-write buffer is full, the predicted last-write blocks must also be scheduled and sent to DRAM. Thus entries in the last-write buffer can be used to hold the next predicted last-write requests.

Given the same group of scheduled write requests, writing them back through condition 1) imposes a smaller penalty to subsequent reads than through condition 2). Tracking last-write blocks using the last-write buffer allows more opportunities to re-distribute the write requests into idle rank cycles. The threshold conditions for the write buffer and the last-write buffer ensure that a large number of scheduling candidates are available to the DRAM controller so they can be scheduled such that they can be efficiently serviced by the DRAM. Details of choosing m and n are given in Section 5.2.

4.2.3 Scheduling Mechanism

When writes are prioritized over reads, the memory controller will schedule a sequence of a maximum number of s write requests to DRAM for service. The memory controller first schedules the row-buffer hit requests for the write with oldest timestamp. If all the row-buffer hit requests for this write have been scheduled, but the number of scheduled requests is still less than s, then the requests to the adjacent banks but same rank will be scheduled. The row-buffer hit and bank-level parallelism requests in the write buffer have high priority to be scheduled over the requests in the last-write buffer. Choosing s is a tradeoff. If we issue fewer, we cause a high bus turnaround penalty and low row-buffer hit rate. If we issue more, the subsequent read requests can be delayed for a long time due to the service for writes. We choose s empirically.

Once the write request in the last-write buffer is ready to issue, it will first search the LLC for that dirty block according to the physical address in last-write buffer. If it is found in the LLC, the dirty block will be pulled from the LLC and send to DRAM for service. Then the corresponding dirty bit for that block will be cleaned. If the block is not found, then it has been evicted from the LLC, so this entry in the last-write buffer will be freed.

Balancing Memory Bandwidth LLC misses tend to occur in bursts. Dirty blocks in or near the LRU position can be evicted in a cluster. These writeback data compete for memory bandwidth with the data being fetched into the LLC, thus degrading system performance [13]. In our technique, the predicted last-write blocks are exposed to DRAM controller once they access the LLC. Exposing last-write blocks to the memory controller at the early stage balances the memory bandwidth, allowing the service of write requests at a time that causes less interference with read requests.

Expanding Write Scheduling Space Write requests in a small scheduling space tend to have low spatial locality. Servicing write requests with low locality imposes a large penalty on subsequent read requests. In our technique, the last-write buffer effectively expands the write scheduling space. The predicted last-write blocks increase the available scheduling candidates. Thus, our technique increases the possibility of scheduling row-buffer hit and bank-level parallelism write requests. Servicing a sequence of write requests with high locality not only improves write service efficiency for DRAM, but also reduces the write-imposed penalty to the subsequent reads.

4.3 Compared with Previous Work

Eager writeback [13] fills memory rank idle cycles with dirty blocks in the LRU position to balance memory bandwidth. But in the eager writeback technique, only write requests in the write buffer can be scheduled by the memory controller. The scheduling space in eager writeback is still limited to the write buffer. Thus, the ability for eager writeback to schedule high locality requests is limited.

The VWQ technique [27] uses the positions near LRU in the LLC as a "virtual write queue." Writes in the virtual write queue are transparent to the memory controller, thereby effectively expanding write scheduling space. But the VWQ technique requires the memory mapping scheme to map the rank, bank and channel bits into the cache index bits of the physical address. Thus, it is expensive for VWQ to be applied to memory mapping schemes that do not satisfy this requirement, such as the mapping scheme in [8, 29]. Both eager writeback and VWQ can be treated as using the recency of the LRU position as a predictor to predict the last-write block. They predict that a dirty block is a last-write block when it comes near the LRU position. Thus, they depend on a policy that divides blocks into distinct levels.

The LWPG writeback technique uses LWP to predict last-write blocks. The LWP is an independent structure and can be applied to any LLC replacement policy. Detecting last-write blocks using LWP enables decoupling the replacement policy from the LLC writeback policy. The LWPG writeback technique can be applied to some inexpensive replacement policies like NRU and random. The NRU policy approximates the recency stack based LRU policy but using only two levels. By contrast, the LRU policy is organized such that each block will reside in a distinct level. Since there can be many blocks in the lower recency level in NRU policy, the recency level can not be an accurate indication for the last-write blocks. The random policy has no information about temporal locality, so no recency information can be used to detect the last-write blocks. Thus, previous work can not be adapted to these simple and light replacement policies. Our technique can detect the last dead blocks without recency information. It completely decoupled with cache replacement policy.

4.4 Storage Overhead and Power

4.4.1 Storage Overhead

In our technique, each core cache keeps a 16 bits partial PC related to each block. For an eight-core 64 KB data cache, it consumes 16KB of storage. In the LLC write simulator, each entry keeps a 16 bits partial PC, 16 bits partial tag, 1 valid bit, 3 bits LRU position. The simulator has 1024 sets and 6 way associativity for a 16M capacity LLC, consuming 27.75KB. The three prediction tables for the skewed dead block predictor are each 4,096 two-bit counters, so they consume 3KB of storage. The last-write buffer has 256 entries per channel, each entry has a 64 bits partial physical address stored in it, it consumes 4K Bytes for a two-channel DRAM system. Thus, the total storage is 16KB+27.75KB+3KB+4KB=50.75KB, which is less than 0.5% of the 16M LLC capacity.

	Leakage Power	Dynamic Power
LWP	0.004 W	0.111 W
LLC	3.674 W	2.405 W
Percent	0.1%	4.6%

Table 1. Dynamic and leakage power evaluation.

4.4.2 Predictor Power

Our technique uses a last-write predictor to predict the last-write cache blocks. We measure the potential impact of this structure on power using CACTI 5.3 [6]. The last-write predictor is composed of an LLC write simulator based on sampling and predictor table. The LLC write simulator was modeled as a tag array of a cache, with only the tag power being reported. The predictor table was modeled as a tagless RAM with three banks accessed simultaneously. In our technique, a PC field is associated with each core cache block. We model this PC metadata as extra data bits in data array.

Table 1 shows the dynamic and leakage power of LWP. As a percentage of the 2.405W total dynamic power of LLC, the LWP uses only 4.6%. Leakage power is always a concern for on-chip power budget. The LWP consumes only 0.1% of the 3.67W total leakage power of LLC. Therefore, the LWP is a reasonably power efficient structure.

5 Evaluation

5.1 Methodology

We use the MARSSx86 [19] simulator together with DRAMSim2 [23] to model an eight-core CMP and DDR3-1600 system. The system configuration is shown in Table 2. We use the SPEC CPU 2006 [5] benchmarks for the evaluation. Of the 29 SPEC CPU 2006 benchmarks, 24 could be compiled and run on our platform. We run six groups of eight-core workloads. Benchmarks are randomly chosen to run in the same run for the six groups. The workloads are shown in Table 3. For each workload, we made a checkpoint by running the one of the memory intensive benchmarks to a typical phase identified by SimPoint [25]. Then we run

Execution core	4.8GHZ, 8 core CMP, out of order, 256 entry reorder buffer, 48 entry load queue 44 entry store queue, 4 width issue/decode, 15 stages, 256 physical registers
Caches	L1 I-cache: 64KB/2 way, private, 64 bytes block size, 2-cycle, LRU L1 D-cache: 64KB/2 way, private, 64 bytes block size, 2-cycle, LRU L2 Cache: 16MB/16 way, shared, 64 bytes block size, 14-cycle, LRU/NRU/random
DRAM and DRAM controllers	2 memory controllers, 1/2/4 ranks per channel, burst length 8, open-page policy 8 banks per channel, 8K bytes row buffer per-bank, DDR3-1600 11-11-11

Table 2. System configuration

Name	Benchmarks
Workload 1	hmmer sphinx3 libquantum GemsFDTD gobmk perlbench lbm astar
Workload 2	perlbench gobmk namd lbm gamess GemsFDTD xalancbmk cactusADM
Workload 3	omnetpp hmmer cactusADM xalancbmk GemsFDTD gcc soplex astar
Workload 4	gromacs astar h264ref lbm omnetpp gcc libquantum calculix
Workload 5	gobmk tonto zeusmp milc bzip2 mcf hmmer astar
Workload 6	omnetpp libquantum hmmer sphinx3 bwaves milc xalancbmk calculix

Table 3. Workloads

the experiment for 2 billion instructions total for all eight cores starting from the checkpoint.

The memory scheduling technique we use for evaluation is first-ready, first-come first-served (FR_FCFS) [22, 21]. Other memory read scheduling algorithms could also work with our writeback optimization; we choose FR_FCFS for simplicity.

We use six writeback optimizations for evaluation. In the graphs that follow, these techniques are referred to with abbreviated names. Table 4 gives a legend for these names.

The write buffer management policy we used for traditional writeback is $rank\_idle\_write$ policy. That is, write requests in the write buffer are scheduled for service when the corresponding rank is idle and the occupancy of the write buffer reaches a threshold, or the write buffer is full. We also evaluated following write buffer management policies: 1) writes in the write buffer are sent to the DRAM only when the write buffer is full, 2) writes in the write buffer are sent to DRAM when the corresponding bank is idle and the occupancy of the write buffer reaches a threshold, or the write buffer is full. Our evaluation shows the $rank\_idle\_write$ buffer management policy yields the best performance for the traditional writeback technique. To ensure fairness we choose to use the $rank\_idle\_write$ buffer management policy for evaluation, but our LWPG writeback technique can be adapted to any buffer management policy.

A large per-channel and per-rank write buffer is complex and power inefficient. Given the same number of write buffer entries for a channel, a write buffer organized by bank consumes less on-chip power because memory read requests only need to search the write entries that target the same bank of the read request. Thus, we evaluate the per-bank write buffer structure with large number of entries, such as 512-entry/c, that is the total number of write buffer entries for a channel is 512 entries. A large number of write buffer entries is complex and space inefficient, thus 512-entry/c per-bank write buffer is the largest write buffer we evaluate. In the LWPG writeback technique, we use a 32-entry/c per channel write buffer and 256-entry/c per rank last-write buffer. We also evaluate the performance of writeback techniques in the presence of prefetching. The result is shown in 5.5.

5.2 Performance Evaluation

We evaluate writeback optimizations with three LLC replacement policies: LRU, NRU and random.

Figure 4 shows the speedups of various writeback optimizations over the baseline in a simulated eight-core processor with LRU LLC and a one-rank memory system; that is, each channel has one rank. For each benchmark we show the speedup of the first run in the random combination.

We choose benchmarks for which the performance of perfect writeback could be improved more than 10% over the baseline. Perfect writeback means all write-induced interference is eliminated. If perfect writeback gives a significant improvement over the baseline for a particular benchmark, that means the performance of this benchmark has a potential to be improved when using writeback optimization. In this experiment, for 16 of 24 benchmarks, the performance of perfect writeback could be improved more than 10% over the baseline. Thus, most of the benchmarks can benefit from writeback optimization in a multi-core system.

From Figure 4, we can see that LWPG writeback technique yields better performance than other techniques. The performance improvement for eager writeback is 4.3% on average over the baseline. The state-of-the-art VWQ technique achieves a 8.1% speedup on average. The LWPG writeback technique yields a average of 8.2% speedup. The

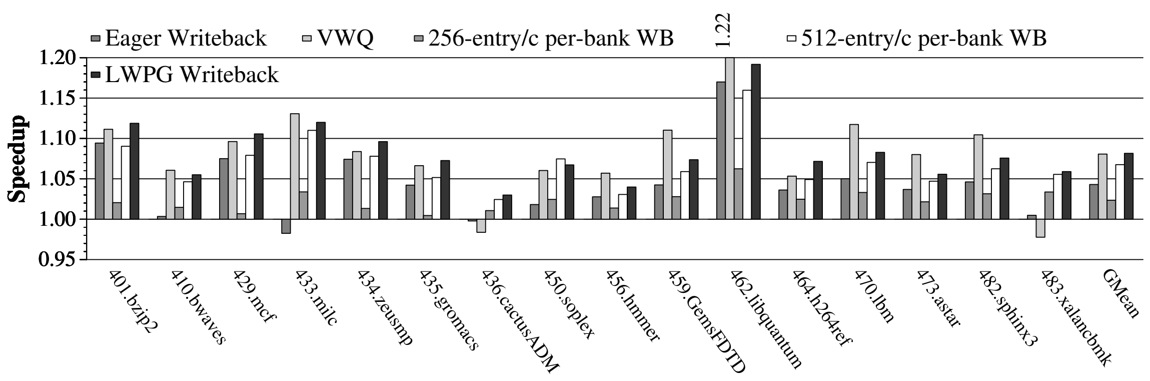


Figure 4. Results running on eight-core one-rank system with LRU LLC

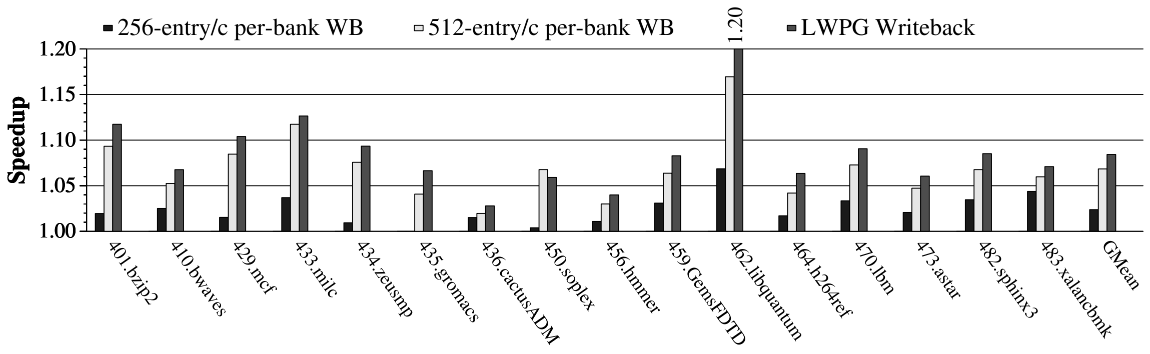


Figure 5. Results running on eight-core one-rank system with NRU LLC

Name	Technique
32-entry/c per-channel WB	Traditional writeback with 32-entry/c per-channel write buffer, this is the baseline
256-entry/c per-bank WB	Traditional writeback with 256-entry/c per-bank write buffer
512-entry/c per-bank WB	Traditional writeback with 512-entry/c per-bank write buffer
Eager Writeback	Eager writeback
VWQ	Virtual write queue
LWPG Writeback	Last-write predictor guided writeback in Section 4

Table 4. Legend for various writeback optimization techniques.

traditional writeback with 256-entry/c and 512-entry/c per-bank write buffer yields 2.4% and 6.8% speedup respectively. Though the 512-entry/c per-bank write buffer has more buffer entries than the LWPG technique, its performance is not as good as the LWPG technique since the per-bank write buffer structure causes conflict misses for write requests that target to the same bank.

Figure 5 shows the IPC speedups with NRU LLC. The NRU recency stack has two levels. The recency information for NRU can not be used to accurately detect the last-write cache blocks. Thus, the eager writeback and VWQ techniques can not be applied to it. The traditional writeback with 256-entry/c and 512-entry/c per-bank write buffer achieve geometric mean of 2.3% and 6.7% speedups respectively. The LWPG writeback technique yields 8.4% geometric mean speedup.

Figure 6 shows the average IPC improvement for one-rank, two-rank and four-rank memory system configurations with LRU, NRU and random replacement policies. The LWPG writeback technique improves performance by 6.5%-11.4% with various DRAM configurations and LLC replacement policies. The system with random LLC replacement policy yields the best performance improvement since the random replacement policy randomly chooses a cache block to be evicted when a new block is placed. Thus, writes in a small write buffer have low spatial locality. The LWPG writeback technique expands the scheduling space, providing more scheduling candidates. For the traditional

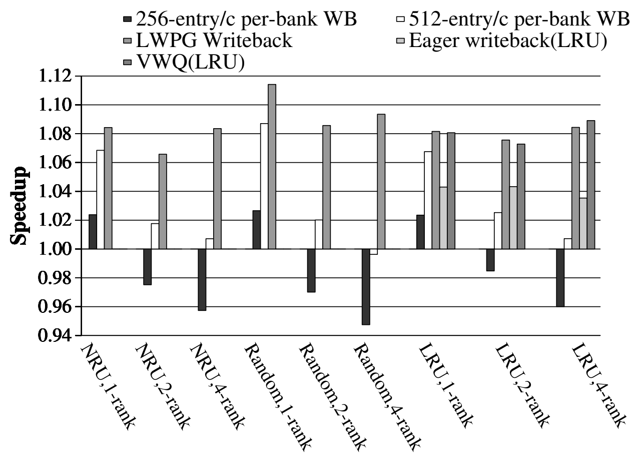


Figure 6. Performance evaluated for various system configurations

writeback with 256-entry/c and 512-entry/c per-bank techniques, the speedups decrease as the number of ranks per-channel increases because increasing the number of ranks per channel decreases the number of write buffer entries for each bank, thus causing more conflict misses for write requests.

In our technique, once the rank is idle and the write buffer has more than m active entries for the idle rank, or the last-write buffer has more than n active entries for the idle rank, a sequence of scheduled write requests will be sent to DRAM for service. Choosing the parameters m and n is a trade-off between the ability to balancing memory bandwidth and expanding the scheduling candidates. Choosing large values for m and n increases the possibility of high locality write requests, but decreases ability to balancing the memory bandwidth. In our experiment, $m = 12, 8, 4$ and $n = 96, 64, 32$ for 1/2/4 rank configurations respectively yields best performance. The maximum number of scheduled requests s each time issued by DRAM controller is also trade-off. A large value of s allows high row-buffer hit rate and low bus turnaround penalty, but can stall pipeline for a long time. In our experiment, we found $s = 12, 16, 16$ for 1/2/4 rank configurations respectively achieves best performance.

5.3 Prediction Evaluation

We evaluate the last-write predictor using false positive rates. The false positive rate is calculated as the number of mispredicted positive predictions divided by the total number of predictions. False positives allow the dirty cache blocks to be written again before they are evicted from the LLC to be written into the DRAM early, thus causing extra memory writes. Figure 7(a) shows that the LWP yields a low false positive rate of 6.6% on average for NRU LLC with one-rank DRAM configuration.

We evaluate the fraction of correctly predicted last-write blocks of LWP. The fraction of correctly predicted last-write blocks is calculated as the number of correctly predicted last-write blocks divided by the number of last-write blocks. A large fraction means more opportunity for optimizations. Figure 7(b) shows the fraction of correctly predicted last-write blocks is 68.8% on average for NRU LLC with one-rank DRAM configuration.

We also evaluate the LWP with all the 1/2/4 rank configurations and LRU, NRU and Random LLC. It yields false positive rate by 6.4%-7.1% and fraction of correctly predicted last-write blocks by 68.8%-76.0% on average with various configurations. This large fraction of correctly predicted last-write blocks and low false positive rates allows more opportunities for optimization without causing significant extra writebacks.

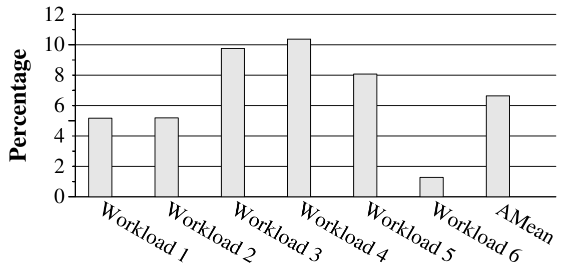
5.4 Row-buffer Hit Rate Evaluation for DRAM Writes

Figure 8 shows results for average write row-buffer hit rates with various configurations. Since caches filter the spatial locality of writes, the traditional writeback with a small write buffer yields low row-buffer hit rate. The 32-entry traditional writeback with a randomly-replaced cache only yields 13.7%-17.3% row-buffer hit rate on average because the random replacement policy randomly chooses a cache block to be evicted once a new cache block comes in. Our technique significantly improves row-buffer hit rate for writes across various configurations to 59.6%-68.6% on average.

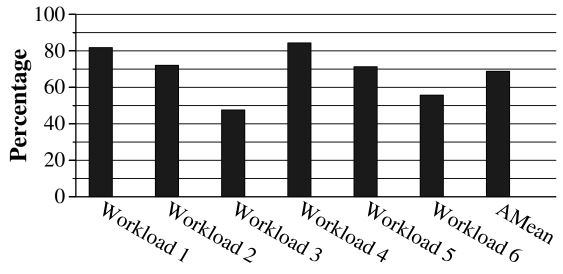
5.5 Performance Evaluation with Prefetching

Hardware prefetching improves system performance by fetching useful data before they are accessed. We evaluate writeback techniques in the presence of prefetching. We model a Middle-of-the-Road Stream Prefetcher [26] with 256 streams. Each LLC request looks up the stream table for issuing eligible prefetching requests. Write-induced interference is a more severe problem in the presence of prefetching because evicted write requests could cause high contention with prefetching requests, thus reducing memory bandwidth utilization.

Figure 9 shows the speedups of various techniques with Random LLC and a one-rank memory system. The baseline technique uses the traditional writeback with 32-entry/c per channel write buffer with the stream prefetcher. The LWPG technique yields better performance improvement over other techniques, achieving a 13.0% speedup on average.



(a) False positive rate



(b) Fraction of correctly predicted last-write blocks

Figure 7. False positive rate and fraction of correctly predicted last-write blocks for last-write predictor with one-rank and NRU LLC configuration

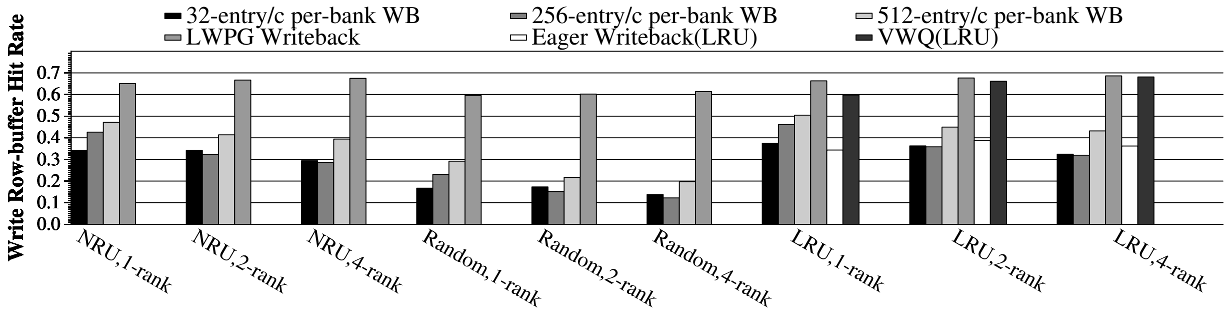


Figure 8. Writes row-buffer hit rate for various configurations

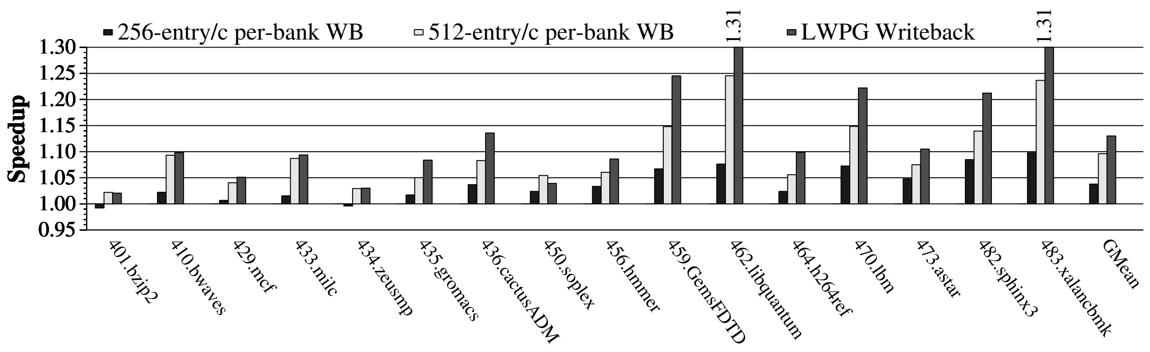


Figure 9. Results running on eight-core one-rank system with Random LLC in the presence of prefetching

6 Conclusion

In this paper, we propose a decoupled last-write predictor guided LLC writeback technique. It uses a last-write predictor to predict last-write blocks in LLC. The predicted last-write blocks are exposed to the memory controller for scheduling. Our technique can balance the memory bandwidth and effectively expands the scheduling space of memory write requests, thus significantly reducing write-induced interference. It is completely decoupled from LLC replacement policy. Our techniques are evaluated for various DRAM configuration by using MARSSx86 simulator together with DRAMSim2. Experimental results show a significant performance improvement over traditional writeback technique.

References

[1] M. Awasthi, D. W. Nellans, K. Sudan, R. Balasubramonian, and A. Davis. Handling the problems and opportunities posed by multiple on-chip memory controllers. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 319–330, New York, NY, USA, 2010. ACM.

[2] S. B.Jacob and D.T.Wang. *The Memory Systems - Cache, Dram, Disk.* Elseiver, 2008.

[3] A. chow Lai, C. Fide, and B. Falsafi. Dead-block prediction and dead-block correlating prefetchers. In *In Proceedings*

of the 28th International Symposium on Computer Architecture, pages 144–154, 2001.

[4] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. High-performance drams in workstation environments. *IEEE Trans. Comput.*, 50:1133–1153, November 2001.

[5] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34:1–17, September 2006.

[6] HP Laboratories. CACTI 5.3. *http: //quid.hpl .hp.com: 9081 /cacti/*.

[7] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the memory system: predicting and optimizing memory behavior. In *Proceedings of the 29th annual international symposium on Computer architecture*, ISCA '02, pages 209–220, Washington, DC, USA, 2002. IEEE Computer Society.

[8] Intel Corporation. Intel 945G/945GZ/945GC/945P/945PL express chipset family datasheet: Intel 82945G/ 82945GZ /82945GC graphics and memory controller hub (GMCH) and Intel 82945P/82945PL memory controller hub (MCH). 2008.

[9] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 39–50, Washington, DC, USA, 2008. IEEE Computer Society.

[10] S. M. Khan, Y. Tian, and D. A. Jimenez. Sampling dead block prediction for last-level caches. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 175–186, Washington, DC, USA, 2010. IEEE Computer Society.

[11] A.-C. Lai and B. Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 139–148, New York, NY, USA, 2000. ACM.

[12] C. J. Lee, V. Narasiman, E. Ebrahimi, O. Mutlu, and Y. N. Patt. Dram-aware last level cache writeback: Reducing write-caused interference in memory system. In *HPS Technical Report*, TR-HPS-2010-002.

[13] H.-H. S. Lee, G. S. Tyson, and M. K. Farrens. Eager write-back - a technique for improving bandwidth utilization. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 33, pages 11–21, New York, NY, USA, 2000. ACM.

[14] H. Liu, M. Ferdman, J. Huh, and D. Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 222–233, Washington, DC, USA, 2008. IEEE Computer Society.

[15] P. Michaud, A. Seznec, and R. Uhlig. Trading conflict and capacity aliasing in conditional branch predictors. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 292–303, June 1997.

[16] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 146–160, Washington, DC, USA, 2007. IEEE Computer Society.

[17] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 63–74, Washington, DC, USA, 2008. IEEE Computer Society.

[18] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 208–222, Washington, DC, USA, 2006. IEEE Computer Society.

[19] A. Patel, F. Afram, S. Chen, and K. Ghose. MARSSx86: A full system simulator for x86 CPUs. In *Proceedings of the 2011 Design Automation Conference*, June 2011.

[20] M. K. Qureshi, M. M. Franceschini, and L. A. Lastras-montao. Improving read performance of phase change memories via write cancellation and write pausing. In *Proceedings of the 2010 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 1–11, 2010.

[21] S. Rixner. Memory controller optimizations for web servers. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 355–366, Washington, DC, USA, 2004. IEEE Computer Society.

[22] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 128–138, New York, NY, USA, 2000. ACM.

[23] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. Dramsim2: A cycle accurate memory system simulator. *Computer Architecture Letters*, PP(99):1, 2011.

[24] J. Shao and B. T. Davis. A burst scheduling access reordering mechanism. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 285–294, Washington, DC, USA, 2007. IEEE Computer Society.

[25] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[26] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 63–74, 2007.

[27] J. Stuecheli, D. Kaseridis, D. Daly, H. C. Hunter, and L. K. John. The virtual write queue: coordinating dram and last-level cache policies. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 72–82, New York, NY, USA, 2010. ACM.

[28] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis. Micro-pages: increasing dram efficiency with locality-aware data placement. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 219–230, New York, NY, USA, 2010. ACM.

[29] Z. Zhang, Z. Zhu, and X. Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 33, pages 32–41, New York, NY, USA, 2000. ACM.