DRAM-Aware Last-Level Cache Replacement

Chang Joo Lee Eiman Ebrahimi Veynu Narasiman Onur Mutlu‡ Yale N. Patt

High Performance Systems Group
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas 78712-0240

‡Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213-3890

TR-HPS-2010-007
December 2010

This page is intentionally left blank.

DRAM-Aware Last-Level Cache Replacement

Abstract

The cost of last-level cache misses and evictions depend significantly on three major performance-related characteristics of DRAM-based main memory systems: bank-level parallelism, row buffer locality, and write-caused interference. Bank-level parallelism and row buffer locality introduce different latency costs for the processor to service misses: parallel or serial, fast or slow. Write-caused interference can cause writebacks of dirty cache lines to delay the service of subsequent reads and other writes, making the cost of an eviction different for different cache lines.

This paper makes a case for DRAM-aware last-level cache design. We show that designing the last-level-cache replacement policy to be aware of major DRAM characteristics can significantly enhance entire system performance. We show that evicting cache lines that minimize the cost of misses and write-caused interference significantly outperforms conventional DRAM-unaware replacement policies on both single-core and multi-core systems, by 11.4% and 12.3% respectively.

1. Introduction

Main memory performance is crucial to high performance microprocessors since resource-limited on-chip caches cannot always store all the data necessary for running applications. Therefore, it is very important to understand the main memory system characteristics (e.g., DRAM characteristics in today's systems) to design high performance microprocessors. In this paper, we make a case for DRAM-aware last-level cache design: we show that designing last-level cache replacement policy to be aware of major DRAM characteristics can significantly enhance entire system performance by minimizing the performance impact of cache misses and writebacks. We consider three major performance-related characteristics of modern DRAM systems that can significantly influence the memory system performance of modern processors: bank-level parallelism, row buffer locality, and write-caused interference.

First, a DRAM chip consists of multiple banks that can be accessed independently. Memory requests to different banks can receive service concurrently. The notion of servicing multiple requests in parallel in different DRAM banks is called DRAM *Bank-Level Parallelism (BLP)*.

Second, data in a DRAM bank can only be accessed from the bank's *row buffer* which essentially serves as a buffer for the last accessed memory row in that bank. Subsequent accesses to the same row can be performed by simply accessing the row buffer, which reduces the latency of the memory access compared to when actually accessing the DRAM cells. This concept is referred to as *row buffer locality*.

Third, write requests interfere with read requests in modern DRAM systems, causing idle cycles on the DRAM data bus. Once a write is serviced, subsequent reads and even some writes (e.g., writes to different rows in the same bank) cannot be started for a certain time even after the write is fully serviced [3]. We call this *write-caused interference*.

Due to DRAM BLP and row buffer locality, different outstanding last-level cache misses may not have the same cost from the processor's point of view. BLP allows the latencies of multiple requests to different banks to overlap, and therefore the processor does not experience each request's memory latency serially. As a result, a request that is serviced concurrently with requests in other banks is less costly than a request that is serviced alone, i.e., with no request in any other bank. Row buffer locality allows requests to the same row

in a bank to be serviced faster. In contrast, the latency of a request that does not "hit" in the row buffer is significantly longer.

Additionally, the eviction of all cache lines does not incur the same cost because of write-caused interference. When a dirty line is evicted by a replacement policy, the modified data must be written back to the DRAM system. The generated write request interferes with read requests that are more critical to the processor's forward progress. As such, interfering write requests should be serviced as quickly as possible to reduce this write-caused interference. Depending on the addresses of outstanding writes, writes can be serviced quickly or very slowly. For instance, multiple writes to the same row in the same bank are serviced very fast (due to row buffer locality), whereas multiple writes to different rows in the same bank are serviced very slowly because they conflict with each other in the row buffer. As a result, to service writes quickly, generating writes to the same row successively is preferable.

Our Observation: As described above, due to DRAM characteristics, not all misses and evictions of the last-level cache incur the same cost. Therefore, it is important for a last-level cache replacement policy to take into account these DRAM characteristics when it makes replacement decisions. However, many previous last-level cache studies mainly focus on deciding which data to store in order to minimize the number of off-chip accesses, solely based on future reuse [1, 16, 2]. A recent replacement policy proposal tries to increase the clustering of last-level cache misses with the hope that they will be serviced in parallel so that the processor does not experience each request's memory latency serially [17], but it does not consider DRAM banks or row buffers. In fact, no previous cache replacement policy explicitly considers the main memory system's characteristics/state to improve overall system performance.

Our goal in this paper is to design DRAM-aware last-level cache replacement policies that aim to minimize the cost of misses and evictions by taking DRAM performance characteristics into account. We propose two policies that work synergistically. The first is a DRAM latency and parallelism-aware replacement policy. The key idea is to favor the eviction of cache lines that when re-fetched will be serviced quickly or in parallel with other misses in the DRAM system. The second is a DRAM write-interference-aware replacement policy. It evicts dirty cache lines that can be written back fast so that writes to DRAM do not interfere with DRAM reads for long periods. Our evaluation shows that the combination of our two policies significantly improves system performance by 11.4% and 12.3% on single and 4-core systems respectively.

Contributions To our knowledge, this is the first paper that proposes last-level cache replacement polices that take into account the characteristics of state-of-the-art DRAM systems. We make the following contributions:

1. We propose a new cache replacement policy that favors the eviction of cache lines that can be brought into the cache quickly with low performance impact in the future (if needed again) because they are likely to hit in a row buffer or likely to be serviced in parallel with other misses accessing different DRAM banks.

2. We propose a new cache replacement policy that favors evicting dirty lines that can be written back to DRAM efficiently by writing to the same row buffer as other outstanding write requests.

3. For both single-core and multi-core systems, we show that each of the proposed policies significantly improves system performance and that the two mechanisms work synergistically. We compare our proposal to the state-of-the-art MLP-aware cache replacement policy that is unaware of DRAM state/characteristics and find that our techniques provide significantly higher performance due to comprehensive awareness of DRAM characteristics.

2. Background: DRAM Characteristics

In this section, we briefly discuss three DRAM characteristics based on the Double Data Rate 3 (DDR3) SDRAM JEDEC standard. We refer readers to the DDR standard documentations and product datasheets [3, 11] for further information. Note that we accurately model all these performance-related timing constraints in our DRAM model for the evaluations described in Section 4.

2.1. Row Buffer Locality

Each DRAM bank is arranged in rows and columns of DRAM cells. The size of a row is several Kbytes (1 $\sim$ 2 Kbytes in each bank per DRAM chip) in modern DRAM systems. To perform a complete access to a data element, three steps are required. First, a precharge command is sent to precharge the bank's bitlines. Second, an activate command is sent to open the source/destination row through the sense amplifier (row buffer) in the bank. Finally, a read or write command is scheduled to access the appropriate column from the row data in the row buffer. Every access can be performed only by reading from or writing into the row buffer. Therefore, if a subsequent access to the bank is mapped to a different row, these three steps (i.e., precharge, activate, and read/write) must be performed again. We call such an access a *row conflict*. On the other hand, a subsequent access which is to the same row as the previous row can be performed simply by accessing the appropriate column from the currently open row. We call such an access a *row hit*. Since a row hit requires only the third of the three steps, its DRAM service time is much less than that of a row conflict.

Figure 1 illustrates exactly how the DRAM system works for these accesses. In Figure 1(a), three reads (A, B, and C) are waiting for DRAM scheduling. They are all mapped to the same row (Row 1) in Bank 0. Currently Row 5 is open in the row buffer of Bank 0. Read A has to go through all three steps since it is a row conflict. The total service time for Read A is the sum of the latencies for the three steps ($t_{RP} + t_{RCD} + CL$), as shown in Figure 1(b). After this latency, the data required by Read A is put onto the data bus. The DDR3 DRAM's *prefetch buffer* allows to enable a burst mode of up to eight (burst length, BL = 8) by bringing (eight) consecutive columns from the row buffer to the prefetch buffer. Therefore eight bursts of data are sent to the data bus. The subsequent two reads can simply access the row opened by Read A. Even though accessing a given column within a row takes only column address strobe latency (CL), consecutive row-hit reads are

serviced even faster. This is because the DDR3 system allows row-hit latencies (CLs) to overlap in order to support back-to-back data transfers among row-hit reads (even among row-hit reads in different banks). Note that such back-to-back data transfers are supported among row-hit writes as well.

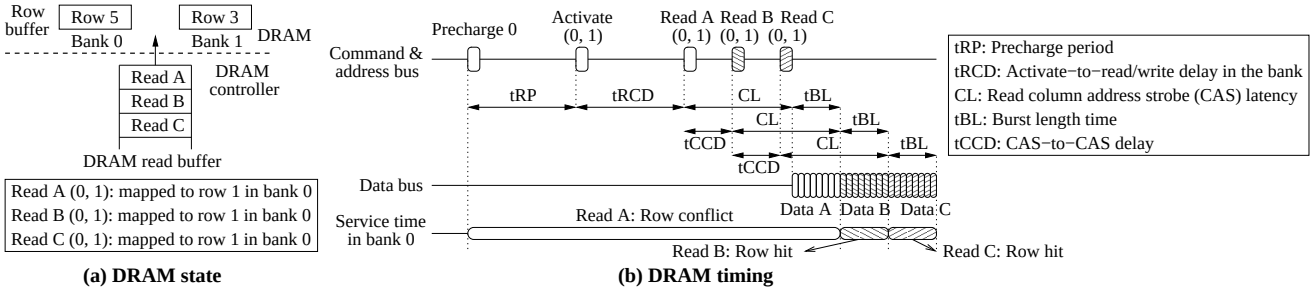


Figure 1. Row conflict and row hit in modern DRAM system

Since row hits can be serviced ($3 \sim 9$ times) faster than row conflicts, many DRAM controllers prioritize row hits over row conflicts in scheduling decisions [18]. To take advantage of this, we propose a cache management policy that takes row buffer locality into account by evicting cache lines that are likely to be row hits rather than lines that would be very costly row conflicts when they need to be re-fetched later.

2.2. Bank-Level Parallelism (BLP)

A DRAM chip consists of multiple ($4 \sim 8$) independent banks and accesses to different banks can be serviced concurrently. Figure 2 shows the DRAM behavior of two row conflict accesses to different banks. Read A is mapped to Row 1 in Bank 0 and the Read B is mapped to Row 1 in Bank 1 as shown in Figure 2(a). Even though they are row conflicts (i.e., the current open rows are different from the rows they access), their DRAM service times can significantly be overlapped as shown in Figure 2(b). Therefore the effective stall time of the processor for these two requests is much less than the sum of the two access latencies. Note that if two row conflicts are mapped to the same bank, they are serviced completely serially and the processor experiences the sum of two row-conflict accesses.[1]

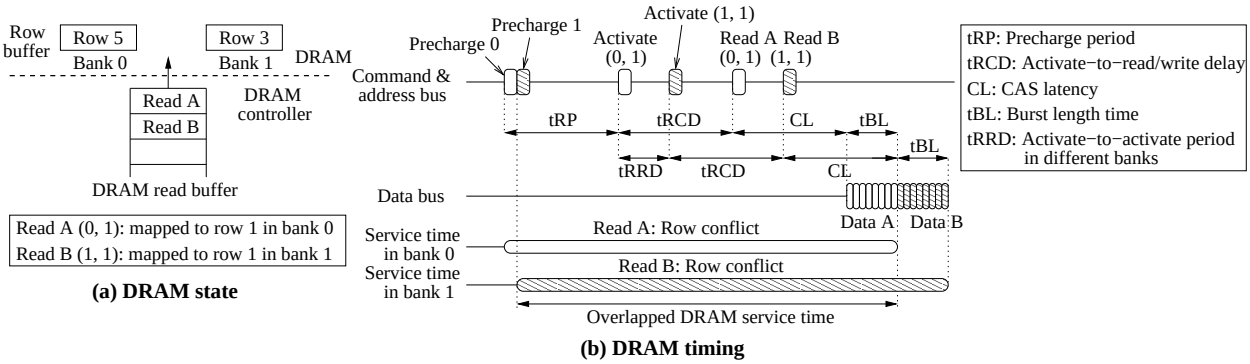


Figure 2. DRAM bank-level parallelism

To exploit DRAM BLP, we propose that the cache replacement policy should favor keeping in the cache

[1]To be precise, the total service time of two consecutive row conflicts in the same bank is more than the sum of two row conflict latencies due to other DRAM timing constraints such as activate-to-activate command period (t_{RC}) and activate-to-precharge command period (t_{RAS}).

those lines whose latencies are unlikely to be overlapped with other concurrent accesses in different banks.

2.3. Write-Caused Interference

Write-caused interference in DRAM comes from read-to-write, write-to-read, and write-to-precharge latency penalties. We first describe read-to-write and write-to-read latencies.

Read-to-write and write-to-read penalties: Read-to-write latency is the minimum latency from a read data burst to a write data burst. This latency is required to change the data bus I/O pins' state from read state to write state. Therefore, during this latency the bus has to be idle. This latency must be satisfied regardless of whether the read and the write access the same bank or different banks. In DDR3 DRAM systems, read-to-write latency is **two DRAM clock cycles**.

Write-to-read (t_{WTR}) latency is the minimum latency from a write burst to a subsequent read command. In addition to the time required for the I/O state change from write to read, this latency also includes the time required to guarantee that written data in the DRAM's prefetch buffer can be safely written to the row buffer (i.e., sense amplifier). Therefore t_{WTR} is much larger (e.g., **six DRAM clock cycles** for DDR3-1600) than read-to-write latency and introduces more DRAM data bus idle cycles. The prefetch buffer is shared by row buffers in all DRAM banks therefore the modified data in the prefetch buffer must be written back to the corresponding bank's row buffer before a read overwrites the data in the prefetch buffer. As a result, write-to-read latency must be satisfied regardless of whether the write and the read are to the same bank or different banks.

Due to read-to-write and write-to-read penalties, switching service between reads and writes frequently in the DRAM system results in many idle cycles. This problem can be mitigated by a write buffer policy [7]. However a write buffer policy cannot solve the problem completely due to the write buffer's limited size and the write-to-precharge latency.

Write-to-precharge latency (write recovery time, t_{WR}) comes into play when a subsequent precharge command is scheduled to open a different row after a write to a bank. This write-to-precharge latency specifies the minimum latency from a write data burst to a precharge command in the same DRAM bank. This latency is very large (**12 DRAM clock cycles** for DDR3-1600) because the written data in the DRAM's prefetch buffer must be written back to the corresponding DRAM row through the row buffer before precharging the DRAM bank. This must be done to avoid the loss of modified data.

Figure 3 illustrates write-to-precharge penalty in a DRAM bank. Write A and Read B access different rows in the same bank (Bank 0). Therefore, after Write A is serviced, a precharge command is required to open the row for Read B (i.e., row conflict). Subsequent to the scheduling of Write A, the precharge command must wait until write-to-precharge latency is satisfied before it can be scheduled. Note that this penalty must be satisfied regardless of whether the subsequent precharge command is for a read or a write. The resulting idle bus cycles is $t_{WR} + t_{RP} + t_{RCD} + CL$ DRAM clock cycles unless there are other requests that are being read

or written in different banks.

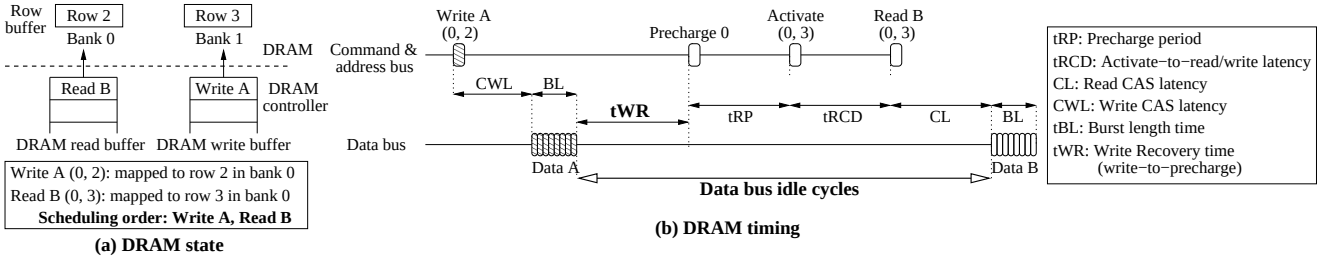


Figure 3. Write-to-precharge penalty in DRAM system

Since the write-to-precharge latency must also be satisfied for a precharge for a subsequent write, row conflicts among writes degrade DRAM throughput for writes. For example, a write to Row 1 after a write to Row 3 in the same bank must still satisfy this write-to-precharge penalty before the precharge command for the write to Row 1 can be scheduled. This problem cannot be solved by the DRAM write buffer. If writes in the write buffer access different rows in the same bank, the total amount of write-to-precharge penalty becomes very large. This eventually results in an even greater delay in the service of reads, thereby degrading application performance.

The source of DRAM writes is dirty line evictions from the last-level cache. A write-caused interference-aware dirty line replacement policy can control the mix of write requests in the DRAM write buffer so that writes can be serviced faster. We describe this policy in Section 3.2.

3. DRAM-Aware Cache Replacement

Our mechanisms aim to minimize the cost of last-level cache misses and evictions of dirty lines by taking DRAM characteristics into account and evicting the least costly cache lines from the last-level cache. We propose two policies: Latency and Parallelism-Aware (LPA) replacement policy and Write-caused Interference-Aware (WIA) replacement policy. We discuss these two mechanisms in the following sections in detail.

3.1. Latency and Parallelism-Aware Replacement

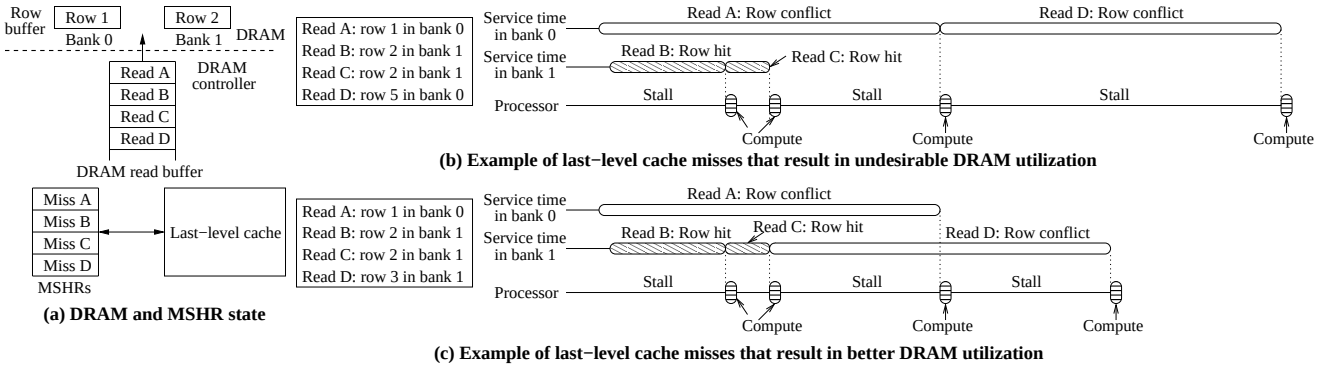
Due to row buffer locality and bank-level parallelism, not all misses incur the same cost from the processor's point of view. The Latency and Parallelism-Aware (LPA) replacement policy favors evicting cache lines that are likely to be row hits or exploit BLP when they are brought into the cache again later.

3.1.1. Why Should We Consider DRAM Characteristics? To answer this key question, we describe the shortcomings of the Memory-Level Parallelism (MLP)-aware cache replacement policy [17] which are due to this technique being unaware of DRAM state/characteristics. The MLP-aware cache replacement policy [17] assumes that clustered cache misses are lower cost than isolated misses. It makes the implicit assumption that the service times of all clustered cache misses are overlapped with each other. Therefore, this policy prefers to evict cache lines that are serviced concurrently with other misses. However, in many cases, concurrent outstanding misses are not necessarily serviced in parallel in the DRAM system. When a large number of

row-conflict misses are outstanding in the memory system, they are serviced in parallel *only if* they are to different DRAM banks. Consider the following example.

Figure 4 describes how the mix of outstanding last-level cache misses can affect DRAM performance and processor stall time. Figure 4 (a) shows the initial DRAM and Miss Status/Information Holding Register (MSHR) state. There are four outstanding misses present in the MSHRs. Row 1 and Row 2 are open in the row buffer of Bank 0 and Bank 1 respectively. Four misses are waiting in the DRAM read buffer to be serviced by DRAM. Figures 4 (b) and (c) demonstrate two scenarios.

Figure 4(b) shows the DRAM service time and processor status when two reads (Reads A and D from Misses A and D) are row conflicts in Bank 0 and two other reads (Reads B and C) are row hits in Bank 1. Since the accesses to Bank 1 are row hits (and therefore low latency), their latencies are overlapped with Read A in Bank 0 (a row conflict). However, Read D is completely serviced alone. The processor must experience the sum of the two row-conflict latencies serially.



(a) DRAM and MSHR state

(b) Example of last−level cache misses that result in undesirable DRAM utilization

(c) Example of last−level cache misses that result in better DRAM utilization

Figure 4. DRAM and processor performance for two different mixtures of outstanding misses

On the other hand, Figure 4(c) shows the DRAM service time and processor status when Read D is mapped to Bank 1 instead of Bank 0 and is still a row conflict (other requests are the same as Figure 4(b)). Read D still takes a long time since it is a row conflict. However, a significant portion of its latency is overlapped with the row-conflict latency of Read A. Therefore this composition of requests results in a significant reduction of processor stall time compared to the previous case.

This example signifies that in contrast to what MLP-aware mechanisms assume, simply having many misses outstanding in the MSHRs does not necessarily mean that those misses are serviced in parallel. Even though Read D is outstanding with three other misses in both Figures 4(b) and (c), its latency is not at all overlapped in the former case yet significantly overlapped in the latter. As such, depending on the mix of clustered misses, their memory service time (or cost) varies significantly.

In addition to isolated misses, clustered misses to different rows in the same bank also incur very high cost. On the other hand, row-hit misses can always be considered low cost due to their low latencies regardless of BLP (recall that multiple row hits' data is transferred back-to-back as discussed in Section 2.1). Rather than simply aiming at clustered memory requests, an intelligent cache control mechanism should be aware of

DRAM characteristics to take advantage of low latency and high parallelism in the DRAM system.

Key insight: To minimize miss cost, a DRAM-aware cache replacement policy can control the mix of requests such that 1) row-hit misses rather than row-conflict misses occur more frequently and 2) row-conflict misses that can be serviced in parallel rather than serially in the DRAM system happen more frequently. Our mechanism does exactly this by measuring these characteristics.

3.1.2. How to Measure DRAM Characteristics Row-hit/row-conflict information can be simply conveyed using one bit in each request from the DRAM controller to the last-level cache. To measure the degree of BLP quantitatively, we define two BLP metrics: 1) *Aggregate BLP* of an application's total execution, and 2) *individual BLP* of a request that is serviced from Cycle N to Cycle M. In the definitions that follow, BLP_i is defined as the number of DRAM banks that are servicing a request in Cycle i. [2] Additionally, $BUSY_i$ is set to one when at least one bank is servicing a request in Cycle i and reset when no bank is servicing any requests.

$$Aggregate\ BLP = \frac{\sum_i BLP_i}{\sum_i BUSY_i} \qquad Individual\ BLP = \frac{\sum_{i=N}^{M} BLP_i}{M - N + 1}$$

Aggregate BLP indicates how many banks were busy servicing requests on average while an application was running. It is greater than or equal to 1 and less than or equal to the total number of DRAM banks. Individual BLP of a request indicates how many banks were busy servicing requests in parallel while the request was being serviced (including the bank servicing that request). Note that these metrics can be measured in the DRAM controller at runtime since the DRAM controller already keeps track of which requests are being serviced in which bank.

3.1.3. Mechanism The Latency and Parallelism-Aware (LPA) replacement policy leverages the observation that if memory requests of an application show high BLP or row buffer locality in a certain execution phase, similar BLP or row buffer behavior will likely occur in the future. For example, current high BLP requests show high BLP when they are refetched later. Previous research [17] also shows that the memory behavior of applications repeats. Therefore LPA assumes that cache lines are *low-cost* if they show high BLP or row buffer locality when they are serviced in the DRAM system. Figure 5 illustrates the logic that performs this function.

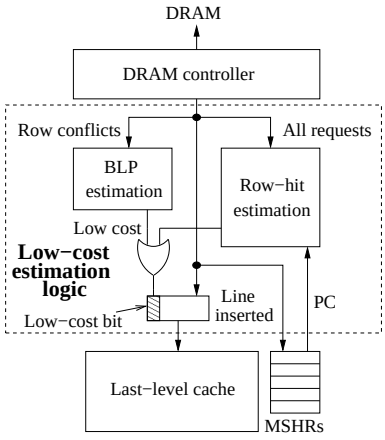


Figure 5. Low-cost estimation for LPA

LPA evicts cache lines that are predicted as low-cost. Low-cost cache lines are identified by a one-bit *low-cost field* in each line. LPA always prioritizes low-cost lines over less

[2] More precisely, a DRAM bank can service multiple row hits at the same time to support back-to-back data transfers as discussed in Section 2.1. However, we assume that only the last request is being serviced in this case to simplify the metric.

recently used lines in the set for eviction. If multiple low-cost lines exist, the least recently used (LRU) line among those is selected as the victim. If there is no low-cost line, the LRU line is evicted.

To take into account temporal locality in reused cache lines, the low-cost bit of a cache line that is reused in the cache is reset. This ensures that LPA performs better than LRU replacement for SPEC benchmarks that work well with LRU replacement. This is because by resetting the low-cost bit of lines that exhibit reuse, LPA retains them in the cache. Additionally, the effective memory latency of misses to low-cost lines that did not exhibit reuse is significantly reduced by taking advantage of row buffer locality and BLP using LPA.

Low-cost estimation using BLP information: To estimate the BLP of a request (or cache line), we need two pieces of BLP information at runtime: the aggregate BLP during a predetermined execution interval of the application and the request's individual BLP. The DRAM controller measures this information and sends it to the estimation logic. Algorithm 1 shows how the low-cost estimation works. The estimation logic works only when the aggregate BLP is greater than $aggregate\_BLP\_threshold$. During a high BLP period, the estimation logic marks as low-cost those requests that had much higher individual BLP ($aggregate\_BLP\_offset$ greater) than the aggregate BLP during that interval.

Starting estimation only when aggregate BLP is high prevents requests from being marked as low-cost during low BLP phases where there is no large performance benefit from BLP. Marking only those requests that show very high individual BLP compared to the aggregate BLP allows the logic to select only those lines for eviction that are likely to exploit high BLP (i.e., it allows the logic to distinguish very low-cost lines from others). We empirically determined the values for $aggregate\_BLP\_threshold$ and $aggregate\_BLP\_offset$.

Algorithm 1 Low-cost estimation using BLP information

```
for each row-conflict request whose service is completed do
  if aggregate BLP > aggregate_BLP_threshold then
    if individual BLP of the request > (aggregate BLP + aggregate_BLP_offset) then
      mark the request as low-cost
    end if
  end if
end for
```

Low-cost estimation using row-hit/row-conflict information: We observe that some application properties can effect the row buffer locality demonstrated in the microarchitecture. The insight here is that the majority of row-hits occur from a few static load instructions. An example is a load instruction that accesses array data elements in a loop. In fact, we find that only 10 static loads are responsible for 65% of all row hits (in the 16 memory intensive SPEC benchmarks shown in Section 4). As such, to estimate whether a cache line is likely to be a row hit, we collect the average row-hit rate of the load instruction that caused the miss.

Algorithm 2 describes how low-cost estimation is performed based on frequent row-hits. We measure the average row-hit rate of a load using a small table (a 16-entry 4-way associative cache structure) each entry of which is associated with a load PC. Each entry keeps track of the total number of requests serviced and the

total number of row hits for the load. Whenever a request is serviced, the table is looked up with the load's PC. If a match is found, its counters are updated as follows: 1) the counter for the total number of requests is incremented, and 2) if the request was a row hit, the counter for the number of row hits is incremented. If no match is found, the LRU entry is replaced with a new entry and its counters are initialized.

Algorithm 2 Low-cost estimation using row-hit/row-conflict information

```
for each request whose service is completed do
  match found ← look up load PC table (request's PC)
  if match found then
    (total number of row hits, total number of requests) ← load PC table (request's PC)
    load PC table (request's PC) ← (total number of row hits + (request row hit ? 1 : 0), total number of requests +
    1)
    adjusted aggregate row hit rate ← MAX(aggregate row hit rate, aggregate_row_hit_rate_min)
    if total number of requests > request_threshold and row hit rate > adjusted aggregate row hit rate then
      Mark the request as low-cost
    end if
  else
    get entry from load PC table (request's PC)
    load PC table (request's PC) ← ((request row hit ? 1 : 0), 1)
  end if
end for
```

Predicting whether a miss is low-cost or not is made using the information looked up from the load PC table precisely before updating the table. If no match is found, the new cache line is estimated as high-cost (i.e., the low-cost bit is not set). If a match is found, the average row-hit rate for the load is calculated by dividing the number of row hits by the number of serviced requests. Prediction is made based on this calculated average row-hit rate and the aggregate row-hit rate for all requests serviced during the corresponding interval.

A fetched line is only considered for low-cost estimation when the row-hit rate information for the corresponding PC is collected for more than $request\_threshold$, not to mark lines for which the corresponding load has only had a few requests serviced. This prevents making a wrong decision about whether the load will likely generate many row hits. The logic marks the line as low-cost only if the row-hit rate of the load that fetched the line is greater than the *adjusted* aggregate row-hit rate for all fetched lines. The adjusted aggregate row hit rate imposes a minimum value of aggregate row-hit rate ($aggregate\_row\_hit\_rate\_min$) to avoid falsely marking lines as low-cost simply because their row-hit rate, although quite low, is larger than a very low aggregate row-hit rate. We empirically determined the set of parameter values used for our evaluation.

3.2. Write-Caused Interference-Aware Replacement

Not all dirty line evictions for the last-level cache incur the same cost. This is because row-conflict writes are much more expensive than row-hit writes as we showed in Section 2.3. The Write-Caused Interference-Aware (WIA) replacement policy's goal is to increase concurrently outstanding row-hit writes. Note that the source of DRAM writes is the last-level cache's writebacks, i.e., dirty line evictions. A write-caused interference-aware replacement policy finds and evicts dirty cache lines that cause row-hit write accesses to

DRAM. To do this it finds dirty cache lines that are mapped to the same row as outstanding writes in the write buffer. The resulting row-hit writes can significantly improve the service time of the writes. The following example shows how such a replacement policy can improve DRAM performance.

3.2.1. Row-Conflict Writes Are Expensive Figure 6(a) shows the initial state of the DRAM read/write buffers and a set of the last-level cache. A row-hit read (Read A) and a row-hit write (Write B) are waiting to be scheduled to DRAM. Two dirty lines (C and D) are at the least recently used (LRU) positions of the shown last-level cache set. Dirty line C is mapped to a different row from Write B whereas Dirty line D is mapped to the same row as Write B.

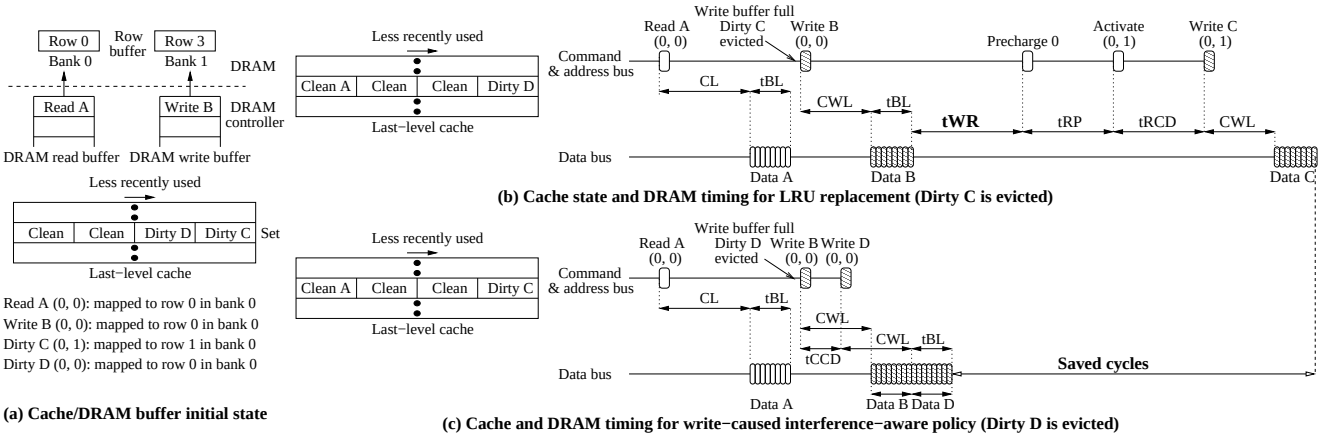


Figure 6. Conventional vs. write-caused interference-aware replacement policies

Figure 6(b) shows the resulting cache state and the DRAM timing when a conventional LRU policy is used in the cache. The LRU line (Dirty line C) is evicted by the fetched line for Read A. Therefore a write (Write C) is generated for Row 1 and is inserted into the write buffer. Writes are serviced in the order of Writes B and C. Because Write C accesses a different row from Write B (row conflict), precharging is required to open Row 1. Since a write was serviced before, write-to-precharge penalty must be satisfied before the precharge command for Write C is scheduled. This increases the idle cycles on the DRAM data bus since the write data for Write C must wait for $t_{WR} + t_{RP} + t_{RCD} + CWL$ cycles after the write burst for Write B.

On the other hand, as shown in Figure 6(c), if Dirty D is evicted instead of Dirty C, the two writes (Writes B and C) are serviced back-to-back, thereby resulting in significant DRAM service time reduction. This example illustrates that a simple cache replacement policy which evicts row-hit writeback requests can improve service time for writes. Our Write-caused Interference-Aware (WIA) replacement policy is designed to achieve this.

3.2.2. Mechanism WIA evicts row-hit dirty lines when a replacement happens in the last-level cache. Ideally, row-hit dirty lines can be found by comparing the row address of each dirty line in the set (which is considered for replacement) with the address of every write in the DRAM write buffer. However, the hardware/design cost of this is not acceptable since it requires an associative search of the write buffer with the address of each dirty line in the cache set. To simplify implementation and hardware cost, we use a row

address register for each DRAM bank to keep track of the address of the last evicted dirty line mapped to that bank. In our address mapping, the last-level set index field includes the DRAM bank index field. [3] Therefore all lines in a set belong to one DRAM bank. This requires one associative search: the stored row address in a register is compared to the address of each dirty line in the cache set. This can be performed by the tag comparison logic in the cache. The tag comparison structure should be modified to support comparing the stored row address with the row addresses of all lines in the set. Figure 7 illustrates this.

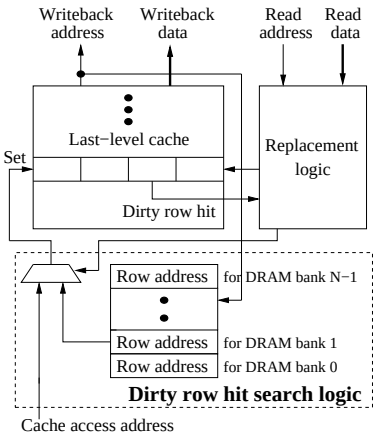


Figure 7. Dirty row-hit search for WIA

Whenever a dirty line is evicted (i.e., a writeback is generated), the corresponding DRAM bank's row address register is updated with the dirty line's row address. When a replacement happens in a cache set, WIA looks for a dirty line that is mapped to the same row as the last-evicted-dirty-line for the corresponding DRAM bank using tag comparison logic in the cache. We found that keeping track of the last evicted row address is enough to gain most of the benefits of searching the row addresses of all writes in the entire write buffer.

WIA prioritizes row-hit dirty lines (if found) over the LRU line for eviction. If multiple row-hit dirty lines are found, the LRU among them is evicted. If none are found, the LRU line is evicted. We found that prioritizing row-hit dirty lines over LRU lines for eviction does not hurt performance due to loss of temporal locality. This is because 1) very few evicted dirty lines by WIA are reused, 2) if the evicted dirty line is required, the write buffer forwards it to the cache unless it is already written back, and 3) performance benefits of evicting row-hit dirty lines outweighs the cost of re-fetching (a small number of) these lines from DRAM.

3.3. Combining Latency and Parallelism-Aware and Write-Caused Interference-Aware Policies

LPA and WIA can be combined to reduce both miss and dirty line eviction penalties. We found that prioritizing row-hit dirty lines (detected by WIA) over low-cost lines (predicted by LPA) for victim decision performs very well. The reasons are as follows.

First, LPA alone is unaware of the dirty line eviction cost. LPA can increase write-caused interference if it evicts dirty lines that cause row conflicts to the same bank, because it only predicts whether or not lines would be low-cost when they are fetched again later. Second, WIA's detection of row-hit dirty lines is more accurate than LPA's prediction of low-cost read misses. This is because WIA looks for dirty lines that can be serviced very soon with other currently outstanding writes, whereas LPA predicts low-cost read misses

[3] This mapping can increase DRAM bank conflicts which in turn causes many row conflicts (among reads and writes with different row addresses). However, a write buffer policy that drains writes only when it is full can mitigate this problem significantly. We use this write buffer policy as presented in Section 4. Also, we find that only keeping track of the globally last evicted dirty line's row address, disregarding which bank it came from, also works well (1% less improvement than the per-bank based approach). This option can be used for systems with different address mapping [22].

that are required in the future. Finally, WIA's penalty of wrong decisions is mitigated by possible forwarding of such cache lines from the write buffer. However, this possibility does not exist with LPA. LPA's wrong decision, evicting a useful and costly cache line, will likely have greater negative affect on performance: the processor must stall for a long time as the cache line needs to be fetched from main memory.

3.4. Multi-Core System Considerations

LPA Replacement in Multi-Core: In many chip-multiprocessors (CMP), multiple cores share the last-level cache and main memory resources. When multiple applications run on different cores, their requests compete with each other for the shared resources. Using *global* BLP and row hit rate (as opposed to per-application information) for low-cost estimation of LPA can cause system performance degradation. There are two reasons. First, this can result in unfair replacement decisions for applications that show high BLP or row buffer locality. Cache lines of such an application that generates many low-cost requests can be unfairly evicted too frequently, which results in performance degradation for such an application. Similarly, cache lines of another application with many high-cost (low row-hit rate and low BLP) misses could be evicted very rarely. Second, global BLP is not repeatable when multiple applications concurrently run on a CMP system. This is because it is not guaranteed that the current phase of application A that is executed concurrently with a phase of application B will be executed concurrently with the same phase of application B again later.

To make LPA effective in CMPs, we estimate low-cost lines on a per-core basis since per-application (or per-core) memory characteristics do not change significantly even on CMP systems. [4] We measure aggregate BLP/row-hit rate and individual BLP/row hit for each core independently. In the definitions of Section 3.1.2, BLP_i of a core is obtained by considering only the banks that are serving that core's requests. $BUSY_i$ of a core is one when at least one request of that core is being serviced in a bank. Aggregate BLP of a core and individual BLP of a core's requests are calculated using these modifications. Low-cost estimation for core A's lines is performed using these aggregate BLP and individual BLP values. In addition, core A's row-hit rate is measured by dividing the number of core A's row-hit requests by the total number of core A's requests serviced in the time interval. Finally, one load PC table is required for each core for low-cost estimation using row-hit information.

When a cache line is inserted into a cache set, LPA determines each core's victim by considering only its lines based on the LPA policy discussed in Section 3.1.3. Among each of the cores' victims, LPA chooses to evict the victim of the core to which the LRU line in the entire cache set belongs.

WIA Replacement in Multi-Core: On the other hand, WIA does not need to be core-aware. This is

[4] Row buffer locality and BLP of an application's requests can be destroyed by other applications' requests in CMP systems. However, we find that due to FR-FCFS (First Ready-First Come First Serve) scheduling [18], row buffer locality is still reasonably stable regardless of other concurrently running applications. Furthermore, if an application's BLP is not broken by concurrently executing applications, we want the cache replacement policy to preserve it. By doing so, FR-FCFS scheduling can still exploit BLP in the memory requests presented to it by the shared cache.

because writes are not critical to an application's progress. Writes become critical only when the DRAM controller cannot service reads due to write-caused interference. Therefore, servicing many writes (from any core) very quickly so that reads (from any core) can be serviced soon and without delay leads to high performance. As such, the WIA policy in multi-core stays the same as we described for single-core systems.

We evaluate our mechanism using these techniques on a 4-core CMP system in Section 5.2.

3.5. Comparison to Memory-Level Parallelism-Aware Replacement Policy

Qureshi et al. [17] proposed an MLP-aware cache replacement policy that prioritizes the eviction of a cache line that is likely to be concurrently serviced with other misses when it is fetched next. Any concurrent misses are assumed to be actually serviced in parallel in the main memory system. This mechanism has multiple important drawbacks compared to our DRAM-aware policies.

First, the MLP-aware policy is not DRAM bank-aware. As we discussed in Section 3.1.1, clustered misses to different rows in the same bank incur very high cost. Since the MLP-aware policy estimates the "MLP cost" of a cache line using the absolute number of outstanding misses (in the MSHRs), it assumes that even misses to the same bank will be serviced in parallel, which is not correct. As such, the MLP-aware policy is prone to mispredicting the cost of misses significantly. Second, the MLP-aware policy does not consider the cost of writebacks. Instead, it considers only the future miss cost of a line when making eviction decisions. This can hurt performance because it can increase write-caused interference in the DRAM system by causing a large number of row-conflict writebacks. As we showed in Section 3.2.1 and empirically evaluate in Section 5, row-conflict writebacks degrade system performance significantly. Third, the hardware/design cost of the MLP-aware policy is more than our proposal. Since MLP cost is stored in each cache line, multiple bits are required in each line (3 bits per line). In contrast, our LPA requires only one bit (indicating low-cost) per line.

4. Methodology

4.1. System Model and Metrics

We use a cycle accurate x86 CMP simulator for our evaluation. Our simulator faithfully models all microarchitectural details such as bank conflicts, port contention, and buffer/queuing delays. The baseline configuration of processing cores and the memory system for single and 4-core CMP systems is shown in Table 1. Our simulator also models DDR3 DRAM performance-related timing constraints in detail as shown in Table 2.

To measure multi-core system performance, we use *Individual Speedup (IS)*, *Weighted Speedup (WS)* [20], and *Harmonic mean of Speedups (HS)* [10]. In the equations that follow, N is the number of cores in the CMP system. IPC_i^{alone} is the IPC measured when application i runs alone on one core of the CMP system (other cores are idle, therefore application i can utilize all of the shared resources) and $IPC_i^{together}$ is the IPC measured when application i runs on one core while other applications are running on the other cores.

Execution Core	4.8 GHz, out of order, 15 (fetch, decode, rename stages) stages, decode/retire up to 4 instructions, issue/execute up to 8 microinstructions; 256-entry reorder buffer; 32-entry load-store queue; 256 physical registers
Front End	Fetch up to 2 branches; 4K-entry BTB; Hybrid branch predictor: 64K-entry gshare/PAs predictor/selector
Caches and on-chip buffers	L1 I/D-cache: 32KB, 4-way, 2-cycle, 64B line size; Shared last-level cache: 16-way, 8-bank, 15-cycle, 1 read/write port per bank, LRU replacement writeback, inclusive, 64B line size, 1, 2MB for 1, 4-core systems; 32, 128 MSHRs, 32, 128-entry L2 access/miss/fill buffers for 1, 4-core systems
DRAM and bus	1, 2 channels (memory controllers) for 1, 4-core systems; 800MHz DRAM bus cycle, Double Data Rate (DDR3 1600MHz) [11], 6:1 core to DRAM bus frequency ratio; 8B-wide data bus per channel, BL = 8; 1 rank, 8 banks per channel, 8KB row buffer per bank;
DRAM controllers	On-chip, open-row, FR-FCFS scheduling policy [18]; 64-entry (8×8 banks) DRAM read/write buffers per channel, drain_when_full write buffer policy

Table 1. Baseline configuration

Latency	Symbol	DRAM cycles	Latency	Symbol	DRAM cycles
Precharge	t_{RP}	11	Activate-to-read/write	t_{RCD}	11
Read column address strobe (CAS)	CL	11	Write column address strobe (CAS)	CWL	8
Additive	AL	0	Activate-to-activate	t_{RC}	39
Activate-to-precharge	t_{RAS}	28	Read-to-precharge	t_{RTP}	6
Burst length	t_{BL}	4	CAS-to-CAS	t_{CCD}	4
Activate-to-activate (different bank)	t_{RRD}	6	Four activate windows	t_{FAW}	24
Write-to-read	t_{WTR}	6	Write recovery	t_{WR}	12

Table 2. DDR3 1600 DRAM timing specifications

$$IS_i = \frac{IPC_i^{together}}{IPC_i^{alone}}, \quad WS = \sum_i^N \frac{IPC_i^{together}}{IPC_i^{alone}}, \quad HS = \frac{N}{\sum_i^N \frac{IPC_i^{alone}}{IPC_i^{together}}}$$

4.2. Workloads

We use the SPEC CPU 2000/2006 benchmarks for experimental evaluation. Each benchmark was compiled using ICC (Intel C Compiler) or IFORT (Intel Fortran Compiler) with the -O3 option. We ran each benchmark with the reference input set for 200 million x86 instructions selected by Pinpoints [15].

Even though we evaluated all the 55 SPEC benchmarks, we report 16 memory intensive benchmarks on which the performance impact of our mechanisms is significant; the effect of our mechanisms on the remaining applications is negligible. Characteristics of the 16 SPEC benchmarks are shown in Table 3. We consider memory read (cache miss) and write (writeback) characteristics independently since LPA is designed for DRAM read efficiency and WIA targets DRAM write efficiency. To evaluate our mechanism on CMP systems, we formed combinations of multiprogrammed workloads from all the 55 SPEC 2000/2006 benchmarks. We ran 17 randomly chosen workload combinations for our 4-core CMP configuration.

4.3. Implementation and Hardware Cost

For evaluations, we periodically measure the aggregate row hit rate and BLP every 100K processor cycles for low-cost estimation. We empirically set $aggregate\_BLP\_threshold$ and $aggregate\_BLP\_offset$ to 2.5 and 0.3 respectively for high BLP estimation. We use a 16-entry 4-way set associative cache structure for the load PC table and set $request\_threshold$ and $aggregate\_row\_hit\_rate\_min$ to 30 and 0.6 for row-hit estimation. BLP and row-hit information required for LPA is collected only from reads (not writes).

Table 4 shows hardware storage cost for our mechanisms on the single and 4-core systems of Table 1. The

			Reads			Writes						Reads			Writes		
Benchmark	Type	IPC	MPKI	RHR	BLP	WPKI	RHR	BLP	Benchmark	Type	IPC	MPKI	RHR	BLP	WPKI	RHR	BLP
179.art	FP00	0.26	90.92	95.43	1.78	9.79	86.75	1.49	482.sphinx3	FP06	0.39	12.94	83.01	1.17	0.63	58.18	1.79
181.mcf	INT00	0.06	107.74	70.08	1.32	11.50	15.03	2.89	171.swim	FP00	0.35	23.10	36.95	2.31	8.24	78.33	2.55
173.applu	FP00	0.93	11.40	90.34	1.56	1.78	81.34	1.74	462.libquantum	INT06	0.67	13.51	94.96	1.01	5.87	89.13	1.06
437.leslie3d	FP06	0.54	20.88	70.50	1.95	2.72	73.80	2.05	481.wrf	FP06	0.72	8.11	72.95	1.47	2.52	76.17	1.70
459.GemsFDTD	FP06	0.49	15.63	45.81	2.21	6.91	50.60	2.70	189.lucas	FP00	0.61	10.61	61.00	1.36	2.38	34.19	1.08
450.soplex	FP06	0.40	21.24	81.64	1.30	3.75	42.48	1.60	436.cactusADM	FP06	0.63	4.51	7.42	1.36	1.22	33.31	1.54
471.omnetpp	INT06	0.49	10.11	63.45	1.27	4.17	6.88	2.46	176.gcc	INT00	0.93	3.24	90.62	1.07	0.54	39.53	1.56
178.galgel	FP00	1.42	4.84	54.45	2.99	1.16	11.51	3.03	464.h264ref	INT06	1.48	1.28	89.56	1.07	0.28	63.55	1.90

Table 3. Characteristics for 16 SPEC benchmarks: IPC, MPKI (last-level cache misses per 1K instructions), WPKI (last-level writebacks per 1K instructions), Aggregate DRAM row-hit rate (RHR), Aggregate DRAM BLP

BLP information (aggregate and individual BLP) is not sent from the DRAM controller to the last-level cache to avoid additional storage and long wires. The BLP estimation is performed in the DRAM controller, and a one-bit field (high/low BLP bit in Table 4) is carried by each request. Similarly, one bit row-hit/row-conflict field is also carried by each request for row-hit estimation before being inserted into the cache.

		Structure	Cost equation (bits)	Single-core's Cost (Bytes)	4-core CMP's Cost (Bytes)
LPA	BLP estimation	Aggregate BLP & busy counters and BLP register	$16 \times 3 \times N_{core}$	6	24
		Individual BLP & busy counters	$16 \times 2 \times N_{bank}$	32	64
		High/low BLP bit	$1 \times N_{buffer}$	4	16
	Row-hit estimation	Aggregate row-hit & request counters and row hit rate register	$16 \times 3 \times N_{core}$	6	24
		Load PC table's tag store (16-entry 4-way)	$27 \times 16 \times N_{core}$	54	216
		Load PC table's data store (row-hit/request counters)	$2 \times 16 \times 16 \times N_{core}$	64	256
		row-hit/row-conflict bit	$1 \times N_{buffer}$	4	16
	Low-cost bit in cache	Low-cost bit	$1 \times N_{line}$	2,048	4,096
WIA		Row address registers	$32 \times N_{bank}$	32	64
Total storage cost for the systems in Table 1				2,250	4,776
Total storage cost as a fraction of the last-level cache capacity				0.2 %	0.2%

Table 4. Hardware cost (N_{core}, N_{line}, N_{bank}, N_{buffer}: number of cores, cache lines, DRAM banks, cache fill buffer entries)

LPA and WIA require only 0.2% of the total last-level cache space on both systems. We assume that the core ID field is already available in each cache line on the 4-core system. If the core ID field (2 bits) is also considered, our mechanisms require 12.7KB (0.6% of last-level cache), which is still insignificant. Note that none of the logic or structures required for the mechanisms is on the critical path.

5. Experimental Evaluation

5.1. Single-Core Results

Figure 8 shows IPC normalized to the baseline LRU for the MLP-aware, Latency and Parallelism-Aware (LPA), Write-caused Interference-Aware (WIA), and combined LPA-WIA replacement polices. The MLP-aware policy is implemented with a set-sampling mechanism that selects between (MLP-aware) linear and LRU policies as proposed by Qureshi et. al [17].

Overall, the best performing policy is the combination of LPA and WIA, which improves performance by 11.4% (6.9% excluding *art*) on average. In contrast, the MLP-aware policy improves performance by 4.6% (0.6% excluding *art*). LPA and WIA complement each other and act synergistically. We make the following major observations:

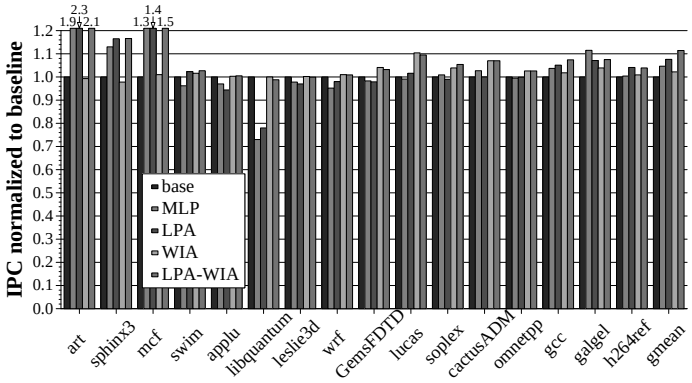


Figure 8. Performance on single-core system

First, both LPA and MLP-aware policies improve performance for *art, sphinx3, mcf, gcc, galgel* and *h264ref*. However, overall LPA outperforms the MLP-aware policy for most benchmarks. Especially, for *swim*, LPA improves performance by 2.3% while the MLP-aware policy degrades performance by 3.8%. The reason why LPA outperforms the MLP-aware policy overall is that LPA is better at identifying and evicting low-cost lines that are serviced faster or in parallel in the DRAM system.

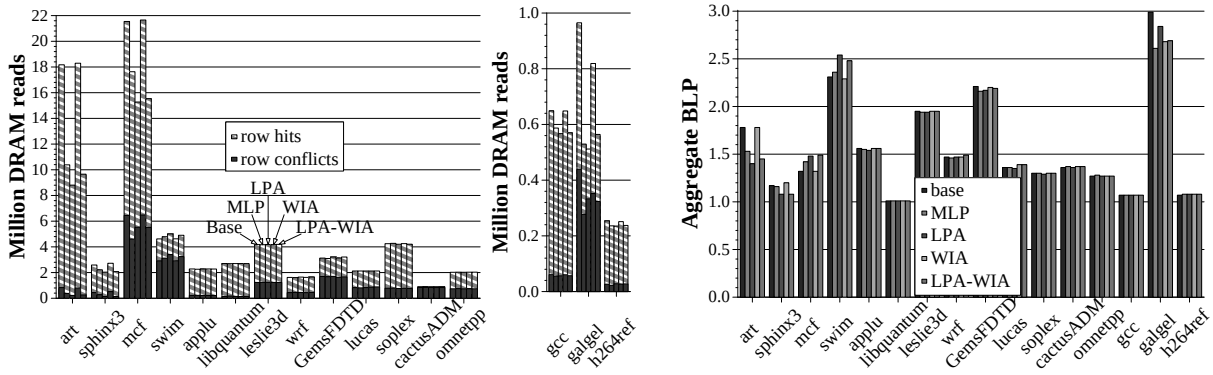
Second, both the LPA and MLP-aware policies degrade performance for *applu, libquantum, leslie3d, wrf,* and *GemsFDTD*. This is because neither are aware of write-caused interference when they evict dirty cache lines. This signifies the importance of write-caused interference when replacement decisions are made.

Third, the performance degradations due to LPA are recovered by employing WIA together with LPA. Additionally, WIA alone improves performance for *GemsFDTD, lucas, soplex, cactusADM*, and *omnetpp* mainly due to its ability in reducing write-caused interference in the DRAM system. As a result, using LPA and WIA (LPA-WIA) together provides the best performance among all policies.

In the subsections that follow we provide further insight using supporting data about DRAM characteristics.

5.1.1. Why Does the LPA Policy Perform Well? Figure 9 shows the total read traffic (from DRAM to the processor) and aggregate DRAM BLP. Read traffic is essentially miss traffic and is divided into row hits and row conflicts. A good cache replacement policy would lead to less read traffic (i.e., fewer misses or higher cache locality), fewer row conflicts, and higher BLP.

LPA reduces row-conflict read traffic significantly for *art, sphinx3*, and *mcf* (by 73.3%, 68.5%, and 14.2% compared to the baseline) in addition to reducing the overall read traffic as shown in Figure 9(a). This improves performance significantly for these applications. LPA significantly reduces overall read traffic for *art* and *mcf* because it mitigates the cache thrashing problem that these suffer from. In such applications, evicting most recently used cache lines improves temporal locality by maintaining at least a portion of the working set in the cache [16]. LPA benefits from this effect since in these applications, the cache lines it predicts as low-cost and chooses for eviction happen to be recently used. The MLP-aware policy also reduces row-conflict read traffic, but much less than LPA.



(a) Traffic (b) Aggregate BLP

Figure 9. DRAM read traffic and aggregate BLP

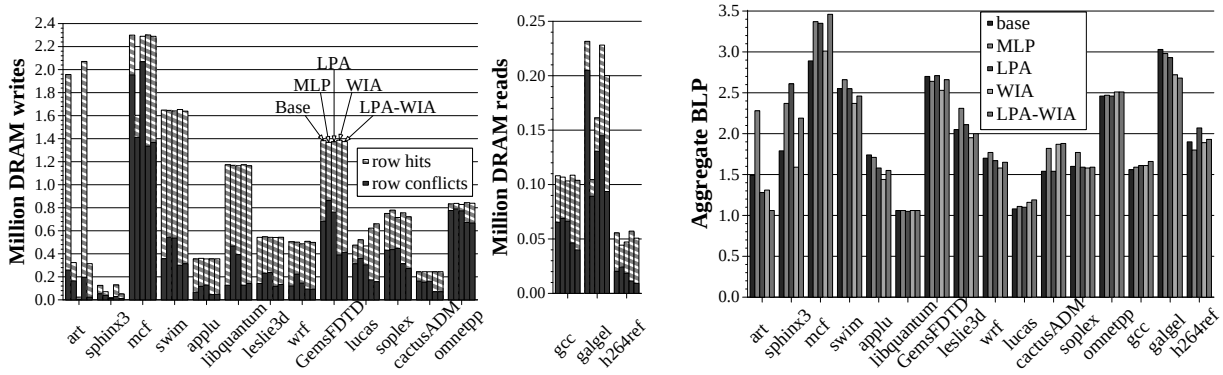
LPA also increases BLP for *mcf* and *swim* by 12.3% and 10.0% compared to the baseline as shown in Figure 9(b). The increased BLP and reduced read traffic cause LPA to outperform the MLP-aware policy. The improved BLP due to LPA translates to performance improvement for *swim* even though LPA increases cache misses (total read traffic) by 8.5%. In contrast, the MLP-aware policy degrades performance of *swim* because many of the concurrent misses it estimates to be low-cost actually end up being high-cost bank conflicts because they map to the same DRAM bank.

LPA significantly outperforms the MLP-aware policy in four applications: *art, sphinx3, mcf*, and *swim*. This is because the MLP-aware policy is not aware of DRAM banks and row buffer locality in the DRAM system. It relies only on the information about how many misses are outstanding at the same time, as discussed in Section 3.5. In contrast, our mechanism explicitly measures and estimates the BLP and row-hit rate in the DRAM system to determine whether a line is likely to be low-cost when re-fetched later.

5.1.2. Why Is Write-Caused Interference Awareness Desirable? Both the MLP-aware and LPA policies degrade performance for *applu, libquantum, leslie3d, wrf*, and *GemsFDTD*, even though the read traffic (i.e., misses or the sum of row hits and row conflicts) and BLP do not change compared to the baseline, as shown in Figure 9(a) and (b). The reason for the degradation can be found by analyzing the write traffic shown in Figure 10(a). Even though the total write traffic does not increase, LPA and MLP-aware replacement policies increase row-conflict writes compared to the baseline. This indicates that these policies increase write-caused interference, causing DRAM performance to degrade due to a large number of idle cycles on the DRAM data bus. In fact, MLP-aware and LPA policies degrade *libquantum*'s performance by 27.0% and 22.0%.

When employed with LPA, WIA reduces the number of row conflicts to as many as the baseline LRU for *applu, libquantum, leslie3d*, and *wrf* as shown in Figure 10(a). It also leads to fewer row conflicts than the baseline for *GemsFDTD*. Hence, by reducing write-caused interference when employed with LPA, WIA recovers the performance degradation due to LPA, and sometimes even improves performance compared to the baseline (for *GemsFDTD* by 3.3%) as shown in Figure 8.

Additionally, WIA alone (without LPA) improves performance for *lucas, cactusADM, soplex*, and *omnetpp*



(a) Traffic (b) Aggregate BLP

Figure 10. DRAM write traffic and aggregate BLP

by increasing row-hit writes (rather than row-conflict writes) compared to the baseline, thereby reducing write-caused interference. Note that for *lucas*, *cactusADM* and *omnetpp*, WIA also increases aggregate BLP for writes, reducing their average latency cost. On the other hand, the MLP-aware policy suffers performance degradation or cannot improve performance for these applications due to its unawareness of write-caused interference in the DRAM system.

5.1.3. Combining LPA and WIA When combined as described in Section 3.3, the performance benefit of each mechanism is obtained additively. This can be justified by observing that the improved DRAM characteristics for reads and writes of each individual mechanism in Figures 9 and 10 do not significantly change for LPA-WIA. We conclude that our DRAM-aware replacement policies significantly reduce costly cache misses and evictions, thereby improving performance significantly on a single-core system.

5.1.4. Effect on System with Prefetching In this section, we discuss the DRAM-aware replacement policy in a system with prefetching. When the DRAM-aware policy is naively employed with prefetching, there are two problems that can reduce its effectiveness. First, useful prefetches that are marked as low-cost by LPA can be evicted (just because they are marked as low-cost) from the last-level cache before they are used. This reduces the effectiveness of prefetching and therefore can hurt performance compared to the baseline LRU policy without LPA. Second, useless prefetches that are not marked as low-cost can stay in the cache for a long time consuming cache space. This can reduce cache efficiency by evicting useful cache lines.

To overcome these problems, we take prefetch usefulness into account in LPA replacement decisions. The basic idea is 1) to ignore the low-cost bit of prefetches that are estimated as useful so that LPA does not evict useful prefetches that are not used yet even if they are predicted to be low-cost, and 2) to evict prefetches that are likely-useless earlier so that cache space can be used for demand and useful prefetches.

We measure prefetch accuracy on an interval-basis. In each interval, if the estimated prefetch accuracy from the previous interval is greater than the *useful_prefetch_threshold*, the low-cost bits of prefetched lines are disregarded by LPA in the current interval. Similarly, when prefetch accuracy is less than *useless_prefetch_threshold*, prefetched lines are prioritized for replacement.

On the other hand, WIA does not require to be prefetch-aware. This is because writes are not immediately critical to an application's progress. Writes become critical only when the DRAM controller cannot service demands and useful prefetches (i.e., reads) due to write-caused interference. Servicing many writes quickly so that reads can be serviced without interruption of writes for a long time leads to high performance.

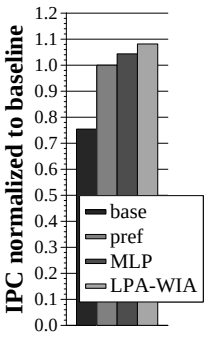


Figure 11. DRAM-aware replacement with prefetching

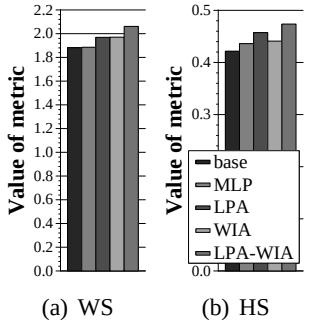
Figure 11 shows the average performance of the baseline with no prefetching, stream prefetching (32 streams, prefetch degree of 4, prefetch distance of 6 cache lines), MLP-aware, and DRAM-aware replacement (LPA and WIA together). We empirically set *useful_prefetch_threshold* as 0.5 and *useless_prefetch_threshold* as 0.2. Prefetch accuracy is measured every 100K processor cycles.

The DRAM-aware replacement policy improves performance by 8.2% compared to prefetching whereas the MLP-aware policy improves performance only by 4.4%. This is because the MLP-aware policy is not aware of DRAM characteristics or prefetch usefulness. We conclude that DRAM-aware replacement is effective in a system with prefetching by taking prefetch usefulness into account.

5.2. Multi-Core Results

We evaluate our mechanisms on a 4-core system with a shared last-level cache. Due to space limitation, we report only average system performance across 17 randomly-selected workloads. Figure 12 shows average weighted speedup (WS) and harmonic mean of speedups (HS) for the baseline LRU, MLP-aware, LPA, WIA, and LPA-WIA replacement policies.

LPA alone improves WS and HS by 4.6% and 8.4% compared to the baseline LRU by evicting low-cost lines while keeping high-cost lines for the application running on each core. WIA alone also significantly improves system performance by 4.7%/4.6% (WS/HS). WIA is more effective in a CMP system than a single-core system. This is because write-caused interference becomes more severe since more reads are generated by multiple cores. Therefore it is more important to service writes quickly so that reads can receive quick service without being interfered-with by writes in multi-core systems. When combined together, LPA and WIA improve WS and HS by 9.5% and 12.3%. On the other hand, the MLP-aware policy improves only HS by 3.4%. Its performance benefit is is small mainly due to its unawareness of DRAM characteristics. We conclude that our DRAM-aware mechanisms are also very effective and improve system performance significantly on multi-core systems.



(a) WS (b) HS

Figure 12. Performance on 4-core system

6. Related Work

To our knowledge, this is the first paper that proposes cache replacement that is aware of the characteristics of the DRAM system. Our approach is to integrate last-level cache and DRAM design tightly. Although there has been significant amount of work in both DRAM and last-level cache management, these mainly considered each component in isolation rather than designing one to be aware of the other.

6.1. DRAM Access Scheduling and Buffer Management Policies

Many DRAM scheduling and buffer management policies [18, 13, 14, 12, 6, 8] have been proposed in previous works. Their mechanisms aim to improve DRAM throughput by taking advantage of row buffer locality and bank-level parallelism in the DRAM system. Our last-level cache policies assume that the underlying DRAM controller exploits these characteristics. Therefore, our mechanisms should work better with a DRAM controller that better exploits the DRAM characteristics. For example, our mechanisms improve system performance (HS) by 13.0% on a 4-core system with parallelism-aware memory scheduling [12].

Most previous DRAM scheduling policies [18, 12, 6, 8] do not address how to manage write-caused interference for high DRAM throughput. In contrast, our write-caused interference-aware (WIA) mechanism evicts dirty lines intelligently from the last-level cache so that overall write-caused interference can be reduced. As such, WIA is orthogonal to these DRAM scheduling and buffer management policies.

Other proposals [9, 13, 19, 7] discuss write buffer management polices and DRAM scheduling for writes. Our baseline employs one of the most recently proposed policies (that does not consider any write for DRAM scheduling until the write buffer is full). For memory intensive applications, this policy is reported to tolerate read-to-write switching penalties better than other alternatives with today's high-bandwidth DDR DRAM systems due to their large write-caused interference [7].

As shown in Section 3, our mechanisms allow the underlying DRAM controller and write buffer management policies to better exploit row buffer locality and BLP for both reads and writes by evicting less costly lines that can exploit those characteristics better.

6.2. Last-Level Cache Management

Many cache replacement/insertion policies were proposed to improve temporal/spatial locality [1, 16, 2]. These are all orthogonal to our work and can be combined with our mechanisms to make them DRAM characteristic-aware.

Jeong and Dubois were the first to propose a replacement policy for a cache with two miss costs (local and remote memory access) [4, 5]. Qureshi et. al. showed that an MLP-aware replacement policy can significantly improve performance by taking into account concurrency level of misses in the memory system [17]. However, none of these policies are aware of DRAM characteristics. We have extensively analyzed and qualitatively and quantitatively compared our proposals to the MLP-aware policy in Sections 3.5 and 5.

Some other studies propose aggressive early writeback policies [9, 7, 21], which proactively send write-backs of dirty lines to the DRAM before they are replaced. These proactive policies also aim to reduce write-caused interference to reads. However, these mechanisms require significant additional hardware and state machines that search for dirty lines to be evicted from the last-level cache. In contrast, our proposal is a simple replacement policy that takes into account the cost of both reads and writes in the DRAM system. As such, our mechanisms can be combined with this prior work to obtain larger benefits than each alone.

7. Conclusion

This paper makes a case for designing the last-level cache policies in a manner that is aware of DRAM characteristics. Previous cache replacement policies overwhelmingly optimize for minimizing cache misses and ignore critical DRAM performance characteristics that affect the cost of each miss: row buffer locality, bank-level parallelism, and write-caused interference. We show that taking these DRAM characteristics into account in the last-level cache replacement policy can significantly improve entire system performance.

Our mechanisms are not limited to DRAM technology-based main memory systems. Other emerging memory technologies are very likely to employ multiple banks and row buffers to provide high bandwidth and low latency. They will also likely impose high write-caused interference due to high bus frequency. As such, the key ideas of our mechanism can be seamlessly applied to emerging memory technologies, and our proposal can possibly be even more beneficial in such systems due to longer read/write latencies.

References

[1] E. G. Hallnor and S. K. Reinhardt. A compressed memory hierarchy using an indirect index cache. In *WMPI*, 2004.
[2] A. Jaleel, K. Theobald, S. C. S. Jr, and J. Emer. High performance cache replacement using re-reference interval prediction (rrip). In *ISCA-37*, 2010.
[3] JEDEC. *JEDEC Standard: DDR3 SDRAM STANDARD (JESD79-3D).* http://www.jedec.org/standards-documents/docs/jesd-79-3d.
[4] J. Jeong and M. Dubois. Optimal replacements in caches with two miss costs. In *Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures*, 1999.
[5] J. Jeong and M. Dubois. Cost-sensitive cache replacement algorithms. In *HPCA-9*, 2003.
[6] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt. Prefetch-aware DRAM controllers. In *MICRO-41*, 2008.
[7] 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 systems. Technical Report TR-HPS-2010-002, The University of Texas at Autin, Apr. 2010.
[8] C. J. Lee, V. Narasiman, O. Mutlu, and Y. N. Patt. Improving memory bank-level parallelism in the presence of prefetching. In *MICRO-42*, 2009.
[9] H.-H. S. Lee, G. S. Tyson, and M. K. Farrens. Eager writeback - a technique for improving bandwidth utilization. In *MICRO-33*, 2000.
[10] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In *ISPASS*.
[11] Micron. *2Gb DDR3 SDRAM, MT41J512M4 - 64 Meg x 4 x 8 banks*. http://download.micron.com/pdf/datasheets/dram/ddr3/.
[12] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *ISCA-35*, 2008.
[13] C. Natarajan, B. Christenson, and F. Briggs. A study of performance impact of memory controller features in multi-processor server environment. In *WMPI*, pages 80–87, 2004.
[14] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *MICRO-39*, 2006.
[15] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *MICRO-37*, 2004.
[16] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high-performance caching. In *ISCA-34*, 2007.
[17] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for mlp-aware cache replacement. In *ISCA-33*, 2006.
[18] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ISCA-27*, 2000.
[19] J. Shao and B. T. Davis. A burst scheduling access reordering mechanism. In *HPCA-13*, 2007.
[20] A. Snavely and D. M. Tullsen. Symbiotic job scheduling for a simultaneous multithreading processor. In *ASPLOS-9*, pages 164–171, 2000.
[21] 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 *ISCA-37*, pages 72–82, 2010.

[22] Z. Zhang, Z. Zhu, and X. Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *ISCA-27*, 2000.