DCA: a DRAM-cache-aware DRAM controller

Cheng-Chieh Huang Vijay Nagarajan Arpit Joshi
Institute of Computing Systems Architecture
University of Edinburgh
{cheng-chieh.huang, vijay.nagarajan, arpit.joshi}@ed.ac.uk

***Abstract*—3D-stacking technology has enabled the option of embedding a large DRAM cache onto the processor. Since the DRAM cache can be orders of magnitude larger than a conventional SRAM cache, the size of its cache tags can also be large. Recent works have proposed storing these tags in the stacked DRAM array itself. However, this increases the complexity of a DRAM cache request, which now translates into multiple DRAM cache accesses (tag/data).**

In this work, we address how to schedule these DRAM cache accesses. We start by exploring whether or not a conventional DRAM controller will work well. We introduce two potential baseline designs and study their limitations. We then derive a set of design principles that a DRAM cache controller must ideally satisfy. Our DRAM-cache-aware (DCA) DRAM controller, that is based on these principles, consistently improves performance over various DRAM cache organizations.

I. Introduction

DRAM caches [1], [2] are beginning to see the light of the day. Intel has announced that its next generation Xeon Phi processor family (code-named Knights Landing) will feature high-bandwidth on package DRAM that can be configured as a cache.

Compared to a conventional SRAM cache, a DRAM cache can be much larger – on the order of hundreds of megabytes or even gigabytes. Consequently, the size of its cache tags can also be on the order of tens of megabytes. Recent works [3], [4], [5], [6], [7], [8] have proposed storing these cache tags in the stacked DRAM array itself. The above works have also proposed a number of different cache organizations and techniques for enhancing DRAM cache performance, either by improving hit latency or by reducing miss penalty.

Storing these tags in the DRAM array, however, increases the complexity of a DRAM cache request. In contrast to a conventional request to DRAM main memory, a request to the DRAM cache will now translate into multiple DRAM cache accesses (tag and data). In this work, we address for the first time the question of how to schedule these DRAM cache accesses. We start by exploring whether or not a conventional DRAM controller will work well in this scenario.

A conventional DRAM controller usually consists of two queues – a read queue and a write queue. Each request to DRAM memory requires only one access to the DRAM array; these read (write) requests are thus stored in the read (write) queue respectively. Switching between read and write modes on the DRAM bus incurs a latency known as *turnaround delay*. Consequently, DRAM controllers strive to avoid frequent turnarounds, as that can be detrimental to overall performance. Furthermore, DRAM controllers typically prioritize reads over writes. This is because read requests tend to be in the critical path of system performance, whereas write requests are usually not.

To Appear in SC16; Salt Lake City, Utah, USA; November 2016

Unlike requests to conventional DRAM memory, *request*s to a DRAM cache will translate into multiple *DRAM accesses*. For example, a read request to a set-associative DRAM cache can translate into three accesses: 1) tag read (to determine hit or miss); 2) data read; and 3) tag write (to update replacement bits). Similarly, a writeback request to the DRAM cache could translate into one tag read and two writes (tag and data). In addition, the DRAM cache also needs to handle refill requests from lower level memory (§II-B2). This added complexity of handling both tag and data accesses increases the challenge of designing a DRAM cache controller.

Study: Conventional Design. We start by exploring a design dubbed *Conventional Design* (CD) – a natural extension of a conventional DRAM controller. In this design, the DRAM cache controller simply pushes accesses into the read or write queue depending on the DRAM access type (read or write). For example, if a DRAM cache read request translates into two read accesses and one write access, the read accesses will be placed in the read queue and write access will be placed in the write queue. Unfortunately, CD suffers from a key limitation — *read priority inversion*. This is because a read access from a DRAM cache read request (high priority) can be blocked by a read tag access coming from a DRAM cache writeback request (low priority). To make it even worse, the two reads (read access from a read request and read tag access from a writeback requests) can potentially generate a row conflict at the DRAM cache – we refer to this as read-read-conflict (RRC).

Study: Request-Oriented Design. To minimize priority inversion and RRC, we consider an alternate design in which the controller pushes accesses into the read or write queue depending on the cache request type (and not the access type). Specifically, all accesses associated with a cache read (writeback) request will be placed on the read (write) queue. We call this the *Request-Oriented Design* (ROD). Unfortunately, we observe that ROD suffers from increased turnaround delays. We also observe that ROD suffers from a longer write queue flushing latency compared to CD. This is because ROD will not schedule read tag accesses for write requests, even when the DRAM cache bus is idle; instead these read accesses will

be scheduled only when the controller starts flushing the write queue. The resulting increased flushing time for the write queue could eventually hurt overall system performance.

Proposed Design. Based on the above observations, we identify a set of principles that a DRAM cache controller should ideally satisfy. The DRAM cache controller should: 1) minimize read priority inversion by taking into account both the access type as well as the DRAM cache request type in scheduling accesses; 2) minimize RRC; 3) minimize the number of turnarounds; and 4) avoid increasing the write queue flushing latency.

We propose DRAM-Cache-Aware (DCA) DRAM controller based on the above principles. Similarly to CD, DCA holds write accesses in a write queue and read accesses in a read queue. However, we categorize read accesses into two groups: priority reads (PRs) and low-priority reads (LRs). Read accesses that are in the critical path – i.e., tag and data reads from cache read requests are classified as PRs. Those read accesses that are not in the critical path – i.e, tag reads from DRAM cache writeback requests and cache refill requests are classified as LRs. Thus, our DCA controller schedules accesses with three decreasing levels of priorities: PRs, LRs and writes. §IV describes how DCA enforces these priority levels in a manner that is consistent with the above identified design principles.

In comparison to CD, DCA ensures that non critical read accesses do not block critical read accesses (coming from read requests) thus mitigating read priority inversion, as well as RRC. In comparison with ROD, DCA experiences reduced flushing times; in addition, turnaround delays are explicitly minimized.

Results and Contributions. Our experiments with a set-associative (direct-mapped) DRAM cache organization on multiprogrammed workloads show that DCA on average is 16.4% (20.8%) faster in comparison with CD, and 7.2% (12.0%) faster than ROD.

We also evaluated the impact of employing a remapping scheme [9], which is an orthogonal technique that has been proposed for minimizing row conflicts in DRAM. We observe that with the addition of remapping (which mitigates RRC), the performance of CD is better than ROD (which continues to suffer from excessive turnarounds). Importantly, even with remapping added, DCA continues to outperform CD by 7% (7.5%) with a set-associative (direct-mapped) DRAM cache organization. This is because DCA is able to mitigate read priority inversion (as well as RRC), whereas remapping is able to mitigate only RRC.

The contributions of this paper are as follows:

- To the best of our knowledge, this is the first study on the impact of the DRAM cache controller on the performance of DRAM caches.
- We establish a conventional DRAM cache controller design (CD) (§III-A) that is based on a conventional DRAM controller. We also introduce an alternate design (ROD) (§III-B) and study the limitations of both designs.
- We propose DRAM-Cache-Aware (DCA) DRAM controller that effectively addresses the limitations of the above designs.

II. Background

In this section, we first briefly discuss the basics of a DRAM controller (§II-A) and DRAM cache (§II-B) that is relevant to this work. We then highlight the potential performance issues (§II-C) that a DRAM cache controller can encounter.

A. Basics of DRAM Controller Design

In a DRAM system, the DRAM bus can be used to service a read or a write request at any given time. Switching the bus between read and write modes is known as turnaround, which incurs a latency known as turnaround delay. Typically, in DDR3-1600, a write to read turnaround (t_{WTR}) will cost 7.5 ns and a read to write turnaround (t_{RTW}) will cost 2.5 ns. Frequent bus turnarounds will result in a performance loss due to these extra latencies. To avoid these turnaround overheads, conventional DRAM schedulers commonly store read and write requests in separate queues – namely, read queue and write queue. Read queue will be served with a higher priority, since read requests are usually in the critical path of system performance. On the other hand, write requests are handled by a passive flushing scheme. The simplest scheme is to service the write queue when the write queue is close to full.

In our work, we consider an optimized scheme which uses two thresholds – a high threshold and a low threshold to determine the flushing point. On reaching the high threshold, the DRAM controller will trigger a forced flush of the write queue. In addition to that, if there are no pending read requests and the occupancy of write queue is greater than the low threshold, the DRAM controller will also flush the write queue. By prioritizing reads over writes, the DRAM controller can avoid turnaround overheads and enhance performance.

B. Basics of DRAM Cache

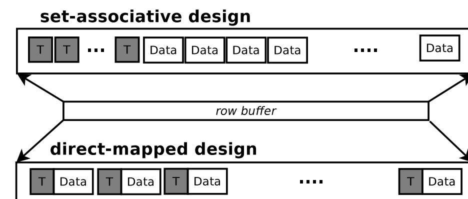

Fig. 1. Different DRAM cache organizations

1) DRAM Cache Organizations: Recent works have shown that a DRAM cache can be organized either as a set-associative cache [6] or as a direct-mapped cache [7]. In both designs, the tags are embedded along with their data in the same row buffer (tags-in-DRAM). However, the relative location of the tag and data are different as illustrated in Fig. 1. In the set-associative design, the tags and the data are segregated; a single cache read requires at least two DRAM accesses – a tag access followed by a data access. On the other hand, the direct-mapped design

tightly integrates tag and data; instead of sequential accesses to tag and data, one could exploit a wider data width to read tag and data in parallel. For further details on these designs, we refer the reader to the work of Loh and Hill [6] and Qureshi and Loh [7].

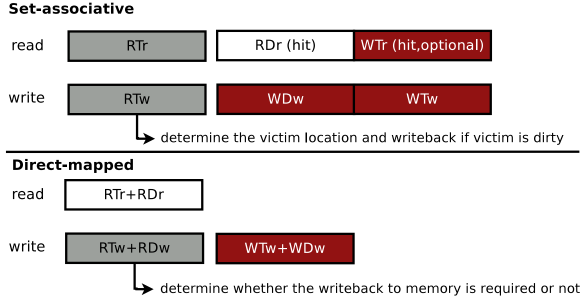

Fig. 2. Accesses in a cache read and a cache writeback.

2) Accesses in DRAM Cache: In conventional DRAM memory, the mapping between a request and a DRAM access is quite simple – a read request translates to a read access and a write request translates to a write access. Compared to conventional DRAM memory, a set-associative DRAM cache requires a more complex mapping. One request typically translates to multiple DRAM accesses. In this section, we describe mappings for three main type of requests in the cache:

Read Request. A read request in a cache will translate into multiple DRAM accesses as shown in Fig. 2: 1) a read tag (RTr) to determine hit/miss and identify the location of the data; 2) a read data to satisfy processor request (RDr); and 3) a write tag to update replacement bits (WTr). Steps 2 and 3 will only happen if the request hits in the cache.

WriteBack Request. A cache write request translates into at least three accesses as shown in Fig. 2: 1) a read tag to obtain the current tag status (RTw); 2) a write data (WDw) and 3) a write tag (WTw). It is worth noting that if the dirty flag of the victim block is set, a read data is also required (RDw).

Refill Request. A cache refill means a block is brought back from the lower level memory and it is waiting to be written into the DRAM cache array. This translation is identical to the write request.

C. Issues with accesses in DRAM cache

Read priority inversion. DRAM controllers have been designed to prioritize processing of read requests over write requests, because read requests are typically in the critical path of execution. DRAM controllers enforce this prioritization by storing accesses corresponding to read/write requests in separate (read/write) queues and prioritizing the processing of read queue over write queue. In a DRAM cache controller, since all requests translate into both read and write accesses to DRAM array, the read queue can potentially contain read accesses corresponding to both read and write requests. In such a scenario, prioritizing the processing of read queue over write queue does not guarantee that read requests will be prioritized over write requests. Indeed, a read access corresponding to a write request can get processed ahead of read accesses corresponding to read requests. This leads to the problem of read priority inversion.

Read-Read-Conflicts (RRC). In the context of DRAM, when a miss in the last level cache also triggers a replacement, the read request because of the miss, and the writeback request because of the replacement of a dirty cache line, can potentially map to different rows in the same bank in DRAM memory. This can lead to a row conflict and this phenomenon is known as Read-Write-Conflicts (RWC) [9]. In the context of a DRAM cache, in addition to row conflicts between read and write accesses, there can also be Read-Read-Conflicts. This is because every writeback request to the DRAM cache (from the upper level cache) consists of a read tag access to determine if the cache block is present in the cache. This read access can conflict with read access(es) from a read request. While write buffering [10] effectively avoids RWC, it does not help in avoiding RRC. On the other hand, a permutation based remapping scheme is able to mitigate RRC in addition to RWC as evidenced by our experiments.

III. Baseline Designs and their Limitations

In this section, we propose two adaptations of a conventional DRAM controller. These designs can be differentiated based on how they classify DRAM accesses corresponding to DRAM cache requests. We then illustrate the limitations of these designs using examples.

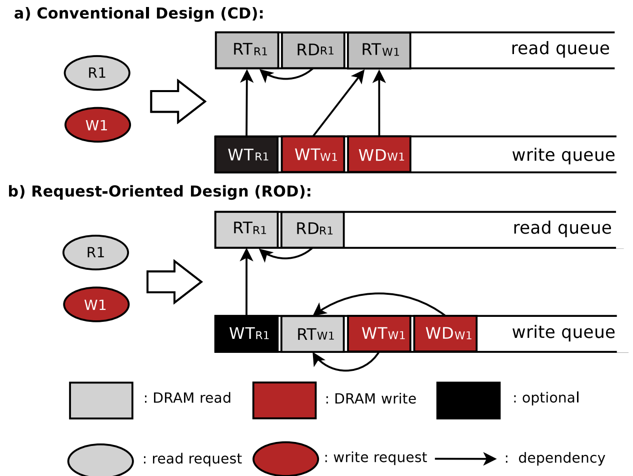

Fig. 3. How the translated accesses map to queues in CD and ROD.

A. Conventional Design (CD)

We first introduce a design that is closely based on DRAM memory controller, and call it *Conventional Design (CD)*. As described in §II-B2, a DRAM cache request requires both read and write accesses to the DRAM array. In CD, DRAM accesses corresponding to a request are classified based on access type. Read accesses will go the the read queue and write accesses will go to the write queue, irrespective of their request type, as shown in Fig.3 (a). This design is very similar to a conventional DRAM controller design as described in §II-A. Using this design, the DRAM cache controller can minimize the frequency of turnarounds.

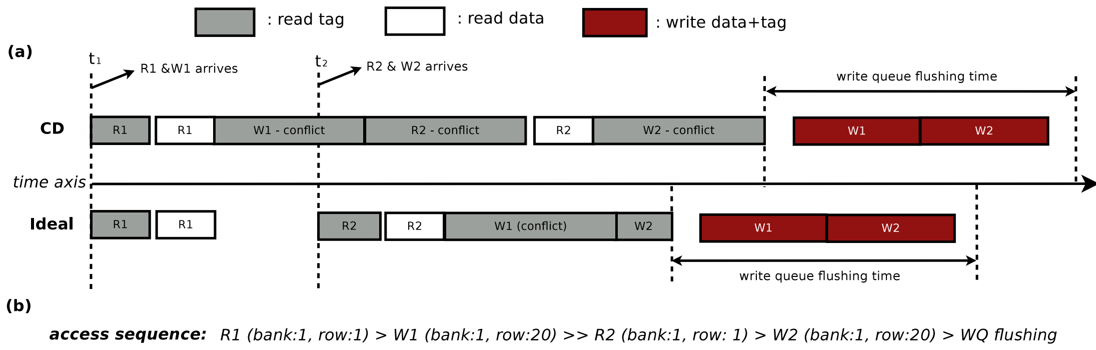

Fig. 4. A case study in conventional design (CD).

However, this design has potential performance issues. As mentioned in § II-C, write requests in DRAM cache generate additional tag accesses that can potentially interfere with accesses from read requests. Since the read queue contains accesses from both read and write requests, read accesses corresponding to read requests compete with read accesses corresponding to write requests. This interference can potentially delay the completion of higher priority read requests. This *read priority inversion* can result in an overall performance degradation, because read requests are usually in the critical path of processor execution. In addition to this, the interference between the read accesses can also result in read-read row conflicts (RRC) as we illustrate next with an example.

Fig. 4 shows an example of how write requests can delay read requests in CD. Consider the sequence of cache requests shown in Fig. 4 (b) (where R* are read requests and W* are write requests). The DRAM cache first receives requests R1 and W1 at time t_1. These requests create three read access entries in the read queue: read tag for R1 (RT_{R1}), read data for R1 (RD_{R1}) and read tag for W1 (RT_{W1}) – and two write access entries in write queue: write data for W1 (WD_{W1}) and write tag for W1 (WT_{W1}). CD will first schedule RT_{R1}. On completion of RT_{R1} it will schedule RD_{R1}. After request R1 is completed, RT_{W1} will be scheduled. As shown in Fig.4(b), because R1 and W1 are accessing a different row in the same bank, RT_{W1} will cause a row conflict and the controller will have to close the previously opened row and re-open another row for request W1. Requests R2 and W2 arrive while the controller is busy handling RT_{W1}. This results in delaying request R2 because it needs to wait for RT_{W1} to complete. After completing RT_{W1}, CD will schedule RT_{R2} and cause another row conflict.

As we can see from this example, CD will trigger three row conflicts in four accesses (RRC). Moreover, request R2 is delayed because it has to wait for RT_{W1}'s completion (read priority inversion). In the figure, we also show an ideal case in which we avoid scheduling RT_{W1} between R1 and R2. Thus, when request R2 arrives it will be scheduled immediately. Also, the ideal case will trigger only one row conflict in this entire request sequence.

B. Alternate design: Request-Oriented Design (ROD)

In this work, we also consider an alternate design, in which DRAM accesses are classified based on their corresponding request type. Accesses corresponding to read requests will go to the read queue and accesses corresponding to write requests will go to the write queue[1] as shown in Fig. 3. We call this design *Request-Oriented Design (ROD)*. Thus both the read and write queues can contain a mixture of read and write accesses to the DRAM array. The advantage of this design is that it will eliminate write interference from delaying read requests.

However, this design also has its own set of limitations. Since the write queue contains both read and write accesses, the frequency of bus turnarounds will increase. This in turn will increase the write queue flushing time. Moreover, the increase in this write queue flushing time can potentially degrade the overall system performance because it can delay subsequent read requests.

Fig. 5 shows an example highlighting this problem. Consider the sequence of cache requests shown in Fig. 5 (b) (where R* are read requests and W* are write requests). The DRAM cache first receives requests R1, W1 and W2 at time t_1. These requests create two read access entries in read queue: read tag for R1 (RT_{R1}) and read data for R1 (RD_{R1}) – and two read and four write access entries in write queue: read tag for W1 (RT_{W1}), write data for W1 (WD_{W1}), write tag for W1 (WT_{W1}), read tag for W2 (RT_{W2}), write data for W2 (WD_{W2}) and write tag for W2 (WT_{W2}). ROD will first schedule RT_{R1}. On completion of RT_{R1} it will schedule RD_{R1}. After request R1 is completed, ROD will not schedule RT_{W1} until the write queue reaches its flushing condition. Request R2 arrives at time t_2. Now, ROD will schedule RT_{R2} and RD_{R2} one after the other. Eventually, ROD starts flushing the write queue. In addition to the writes, RT_{W1} and RT_{W2} also need to be processed. This results in a longer write queue flushing time.

As we can see from this example, ROD fails to utilize the idle time between processing requests R1 and R2 to schedule RT_{W1} and RT_{W2}. Whereas in the ideal case shown in the

[1] With one exception: the write tag for a read request (if present) would go to write queue for performance reasons.

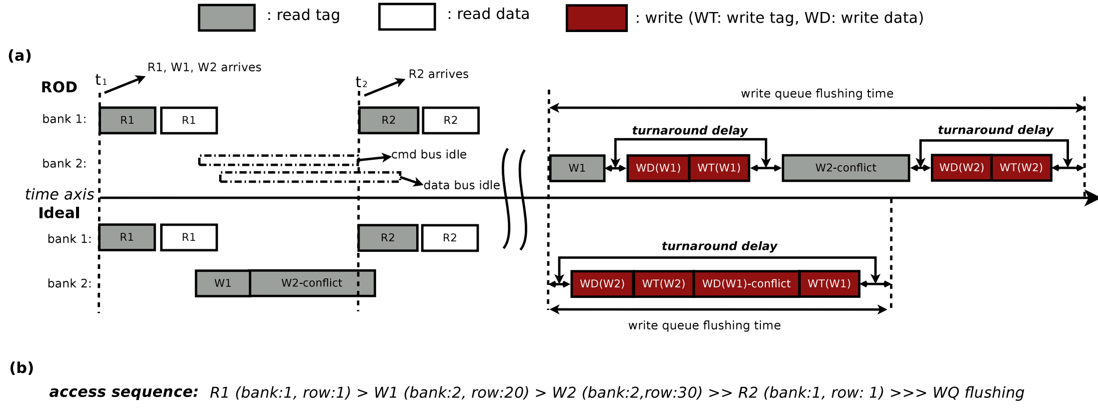

Fig. 5. A case study in Request-Oriented Design (ROD) — please note that bank 1 and bank 2 are in the same rank/channel.

DRAM-Cache-Aware:

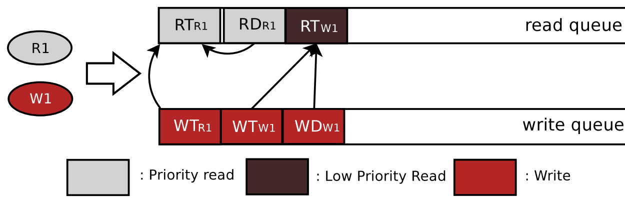

Fig. 6. How the translated accesses map to queues in DCA design

figure, read tags for write requests are scheduled opportunistically and the overall completion time of the sequence is lower.

IV. DRAM-Cache-Aware DRAM Controller

In this section, we first propose a set of design principles for designing a DRAM cache controller. We then propose the DRAM-Cache-Aware (DCA) DRAM controller which is designed based on those principles.

A. Design Principles

Based on our observations in §II, we propose a set of principles for designing a DRAM cache controller as follows:

- Avoid priority inversion by taking into account both the access type as well as request type in prioritizing accesses (§II-A) – this can avoid the scenario where priority reads are delayed by low-priority reads.
- Avoid Read-Read-Conflicts (§II-C) – RRC can potentially cause row conflicts and degrade performance.
- Minimize turnarounds (§II-A) – frequent turnarounds can result in performance degradation.
- Avoid increasing the write queue flushing latency (§III-B) – if the time to flush the write queue is increased, subsequent read requests can be delayed.

B. Handling Low Priority Reads

In our DRAM-Cache-Aware DRAM controller, we first classify read accesses in two categories: priority reads (PR) and low priority reads (LR). *Read accesses corresponding to cache read (CR) requests are classified as PR* and *read accesses corresponding to cache write (CW) requests (including cache refill requests) are classified as LR.*

In this work, we propose a scheduling scheme to handle these PRs and LRs. In our DRAM-Cache-Aware scheme, the translated accesses will map to the read/write queues in the same way as in CD. However, the scheduler will differentiate between PRs and LRs in the read queue (as shown in Fig. 6). In a normal scheduling slot, the scheduler will only schedule PRs and hold the LRs in the queue (like a write queue). The LRs in the read queue will only be scheduled under two conditions: either the occupancy of read queue reaches a pre-defined threshold, or the LRs satisfy a criteria that is specified in §IV-C. Algorithm 1 shows the detailed steps of DCA scheduling scheme. It is worth noting that DCA is based on a recently proposed application-aware scheduling algorithm – BLISS [11]. However, our scheme is not limited to any scheduling algorithm.

Algorithm 1 DRAM-Cache-Aware Scheme

```
1: //ScheduleAll is initialized to False
2: if (ReadQueue.occupancy ¿ 85%) then
3:     ScheduleAll = True
4: else if (ReadQueue.occupancy ¡ 75%) then
5:     ScheduleAll = False
6: end if
7:
8: //done = False implies that no reuqest is ready
9: if (ScheduleAll) then
10:     done = BLISS_ScheduleAll(readqueue)
11: else
12:     done = BLISS_SchedulePROnly(readqueue)
13: end if
14:
15: if (!done AND !ScheduleAll) then
16:     //no PRs in the readqueue
17:     OFS_Flush(readqueue)
18: end if
```

C. Flushing Scheme for Low Priority Reads

When to Flush? When to flush the LRs is an important design decision. If we only flush LRs when the queue is full (or close to full), then our design might lose the opportunity to utilize the available DRAM bandwidth. On the other hand,

if we schedule LRs too aggressively, then these LRs could cause priority inversion and RRC issues as described in §II-C. Therefore, in this work, we strive to strike a balance. In principle, we would like to schedule LRs along with the PRs, as long as there is no interference between them. Here we observe that RRCs typically appear due to the spatial locality present in DRAM accesses. Therefore, RRCs can be avoided by not scheduling those LRs that are accessing recently accessed banks.

Obtaining the access footprint for each bank. In order to let the scheduler determine the recency of accessed banks, we use a technique similar to the one used to implement cache replacement policies. We use a 3-bit bank re-reference prediction counter (RRPC) to keep track of the bank re-reference history and use it to determine (predict) whether or not we should schedule a particular LR request. This RRPC design is similar to a cache replacement technique called *re-reference interval prediction (RRIP)* [12]. In our proposed design, each bank will have a 3-bit RRPC counter which amounts to 24 bytes overhead for a DRAM organization with 64 banks. This counter will only change when a PR is accessed. Initially, the counter is set to 0. When there is a PR request to the DRAM controller, the system will first decrease the counter in all banks by 1 (0 will stay at 0) and set the most recently accessed bank's counter to 7. By doing this, we can determine the recency of each bank. It is worth noting that the RRPC will only be modified for PRs.

Opportunistic Flushing Scheme (OFS). When there are no pending PRs and the occupancy of the read queue is below the flushing threshold, OFS can start scheduling LRs. When the scheduler is ready for scheduling a LR (say request LR_0), we first check if there is a row conflict in the corresponding bank. If there is no row conflict (either row buffer hit or a closed row), LR_0 will be scheduled. On the other hand, if there is a row conflict in that bank, we will check the corresponding bank's RRPC. If the RRPC is lower than a pre-defined threshold that we call *flushing factor* (FF), our controller will schedule LR_0. However, if the request can not meet any of above criteria, it will not be scheduled and will have to wait for next available scheduling slot for LRs. In our study, we found that the design is not very sensitive to flushing factor when it is smaller than 5 (FF-5). In our multiprogrammed workloads, the average performance difference from FF-4 to FF-1 is less than 1%. Therefore, in this work, *we use FF-4 as the flushing factor*.

D. Summary

In this section, we explain through an abstract example how DCA can improve performance compared to other policies. In this example, let us assume that the DRAM cache controller receives 5 cache read requests (CR) at different time instances. These CRs are classified as priority reads as discussed in §IV-B. Moreover, since read requests are typically in the critical path of system performance, delaying the completion of a read request might delay the arrival of a subsequent read request. Meanwhile, cache write requests also arrive in the controller and are translated into reads (for tag) and writes (for tag+data). At the time of completion of the fourth CR, the write buffer is nearly full and this triggers the flushing of the write queue. At this moment, any incoming CR cannot be served before the write queue occupancy returns below the low threshold. As shown in Fig. 7, CD takes more time to service the first 4 CRs due to the interference of the reads from CW and results in a long time being taken to complete the five CRs. In ROD, we buffer all CWs in write queue and complete them when write queue flush is triggered. As mentioned in §III-B, write queue flushing will take longer in ROD and this delays servicing of the fifth CR. Compared to them, DCA will classify reads from CW to low priority reads (LRs) and prioritize CR over LR in the read queue as long as the read queue occupancy is below the LR queue flushing threshold. Meanwhile, OFS will try to schedule LR when there are no CRs pending and RRPC is lower than the flushing factor.

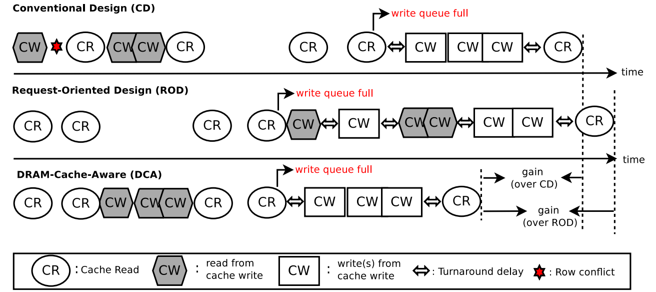

Fig. 7. A comparison among CD, ROD, and DCA.

1-2	soplex-mcf-gcc-libquantum	astar-omnetpp-GemsFDTD-gcc
3-4	mcf-soplex-astar-leslie3d	bwaves-lbm-libquantum-leslie3d
5-6	omnetpp-milc-leslie3d-astar	soplex-astar-lbm-mcf
7-8	lbm-omnetpp-leslie3d-bwaves	milc-leslie3d-omnetpp-gcc
9-10	bwaves-astar-gcc-leslie3d	omnetpp-libquantum-mcf-gcc
11-12	gcc-libquantum-lbm-soplex	gcc-leslie3d-GemsFDTD-soplex
13-14	lbm-libquantum-omnetpp-bwaves	gcc-mcf-leslie3d-milc
15-16	omnetpp-mcf-leslie3d-lbm	libquantum-lbm-soplex-astar
17-18	milc-libquantum-bwaves-GemsFDTD	leslie3d-astar-libquantum-bwaves
19-20	lbm-gcc-mcf-libquantum	soplex-astar-GemsFDTD-leslie3d
21-22	GemsFDTD-astar-leslie3d-libquantum	libquantum-milc-lbm-mcf
23-24	lbm-libquantum-leslie3d-bwaves	milc-leslie3d-omnetpp-bwaves
25-26	bwaves-astar-GemsFDTD-leslie3d	gcc-soplex-libquantum-milc
27-28	omnetpp-lbm-leslie3d-GemsFDTD	soplex-bwaves-GemsFDTD-leslie3d
29-30	GemsFDTD-leslie3d-libquantum-milc	omnetpp-bwaves-leslie3d-GemsFDTD

TABLE I
WORKLOAD GROUPINGS

V. Experimental Methodology

We use gem5 [13], a cycle-accurate simulator for our experimental evaluation. We model an OoO x86 core based CMP system with private L1 and shared L2 caches. The processor is able to issue up to 8 instructions per cycle and has 192 ROB entries. Using memory-intensive benchmarks from SPEC 2006 [4], [14], we generated 30 4-core workloads for evaluating the controller designs. The details about these workloads are shown in Table I. For our simulations, we fast-forward 4 billion instructions and simulate timing model for the subsequent 500 million instructions. It is worth noting that DRAM cache is huge and takes relatively longer time to warm up, so caches (w/o timing) are also simulated during fast-forwarding. For all speedup results in this paper, we use the

System Parameters	
Processor	4GHz, x86, 192 ROB Entries, 8-wide
L1 I/D caches	each 32KB/2way, 2 cycles, private
L2 cache	8MB, 20 cycles, shared
L3 cache	DRAM Cache, 256MB (240MB data capacity), 1/15 way
On-chip bus	4GHz, 256-bit width
Off-chip bus	2GHz, 64-bit width
Memory latency	50ns

Die-Stacked DRAM	
Timing Params	tRCD-tCAS-tRP-tRAS 8-8-8-30 (ns) tWTR-tRTP-tRTW 5-7.5-1.67 (ns) tWR-tBURST 15-3.33 (ns)
Organization	16 banks/rank, 1 rank/channel 4 channels, 4KB row buffer, RoBaRaChCo, open-page
Read Queue	64 (32 for ROD) entries per channel Flush thres.: 75%/85% (for DCA scheme.) BLISS [11]
Write Queue	64 (96 for ROD) entries per channel Low/high flush thres.: 50%/85% BLISS

TABLE II
SYSTEM AND DIE-STACKED DRAM PARAMETERS

normalized weighted speedups [15] and for reporting average speedup, we use geometric mean.

The detailed system parameters and stacked DRAM parameters that we use are shown in Table II. Following [6], [7], [14] we simulate a $256MB$ non-inclusive DRAM cache using a detailed DRAM timing model. Since DCA only focuses on improving the scheduling of DRAM cache accesses, as opposed to improving the DRAM cache hit rate, its efficacy is not sensitive to the cache size. With die-stacked DRAM latency not improving to the expected extent [16], we use timing latencies similar to Sim et al. [14] which is half-way between today's latency [17] and the predicted latency [18].

Conceptually, die-stacked DRAM is a wide-io DRAM chip stacked on the processor chip. According to JEDEC [17], [19], the turnaround delay refers to *the delay from start of internal write/read transaction to internal read/write command*. This latency has remained fairly unchanged across multiple DDR generations. So, we believe it should not be affected significantly because of die-stacking. In addition, according to JEDEC wide-io DRAM standard [19], the *minimal* tWTR in wide-io DRAM will be 2 tCK (10 ns), which is higher than desktop's DDR3-1600 (7.5 ns) [17]. In this work, we conservatively assume that wide-io DRAM timings could be improved in the future [16] and use a tWTR latency of 5 ns which is twice as fast as that of JEDEC standard's 10 ns. It is worth noting that this conservative assumption will only lower the speedup of our design over ROD.

Since a DRAM cache could be organized as either a direct-mapped [7] or a set-associative cache [6], we evaluate DCA for both of these designs. Recall that the major difference between these two designs is the number of accesses per DRAM cache request (as shown in Fig. 2). We evaluate our proposed DCA controller and compare it with the conventional design (CD), and the request-oriented design (ROD). In each of the above designs, we use MAP-I [7] as the DRAM cache miss predictor for reducing miss penalty. The underlying scheduling algorithm for all the designs in our study is BLISS [11].

VI. RESULTS

In this section, we first analyze the overall performance improvement (§VI-A) of the proposed DCA design in comparison with CD and ROD (for both set-associative and direct-mapped organizations) using multiprogrammed workloads. We further highlight the improvement in L2 miss latency (§VI-B) which helps in explaining the improvement in overall performance. We finally highlight the magnitude of turnaround overhead (§VI-C) and the variation in row buffer hit rate (for read accesses) across the different designs. This analysis helps in explaining the characteristics of different designs.

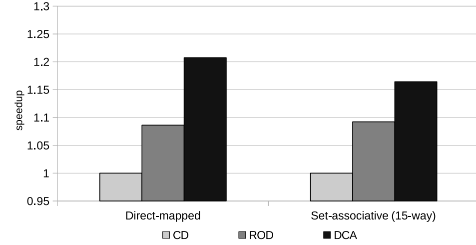

Fig. 8. Average performance speedup

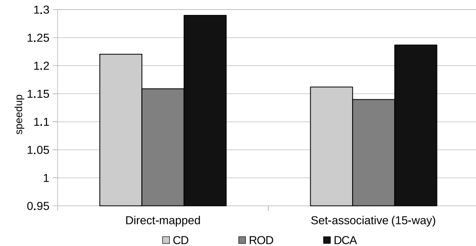

Fig. 9. Average performance speedup with remapping [9]

A. Performance

Fig. 8 shows the normalized weighted speedup of all the proposed designs, for both set-associative and direct-mapped organizations, averaged across all workloads. All speedups are normalized to CD. Among all the designs, we can see that CD has the worst performance. This is because CD suffers from issues that we introduced in §II-C. ROD on the other hand achieves an improvement of 8.6% and 9.2% for direct-mapped and set-associative caches respectively. ROD is able to achieve this improvement because it reduces the impact of priority inversion and RRC by naively scheduling DRAM accesses according to the cache request type. However, it is important to note here that ROD suffers from a high turnaround overhead (which we will discuss in §VI-C). The proposed DCA design has the best speedup for both the cache organizations and achieves 20.8% and 16.4% improvement for direct-mapped and set-associative caches respectively. DCA is able to achieve this improvement because it not only reduces the impact of priority inversion and RRC, but also minimizes the overhead of turnaround delay. Overall, we can see DCA provides more speedup in the direct-mapped design. This is because a single read request in a set-associative cache requires more read queue entries (§II-B2). Since we use the same read

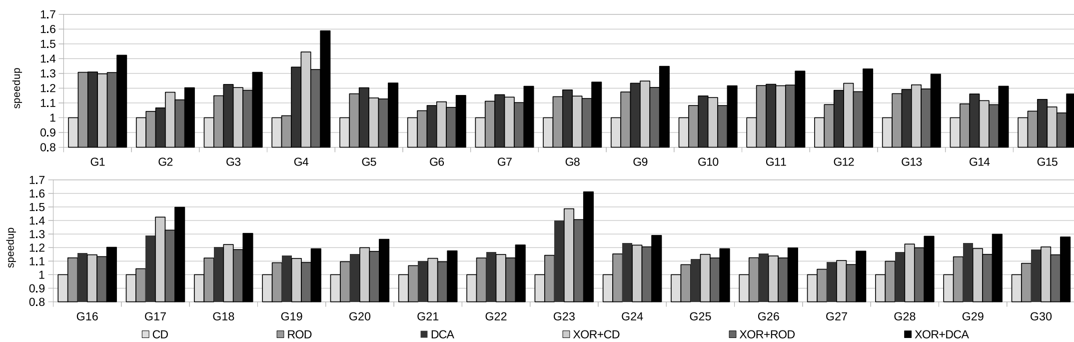

Fig. 10. Performance speedup of all designs (set-associative)

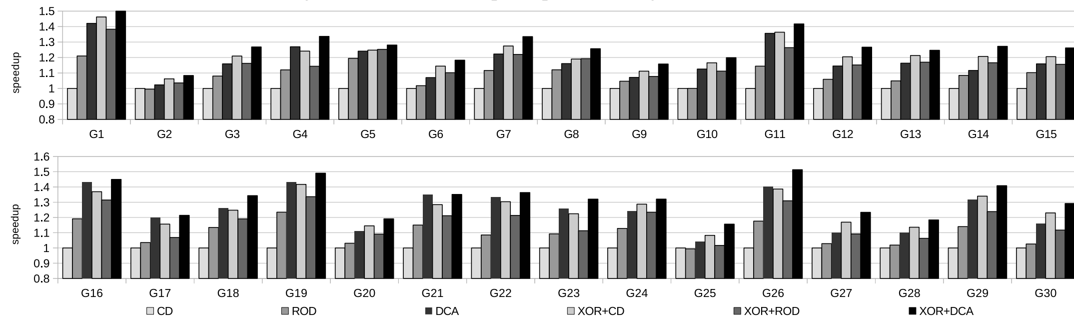

Fig. 11. Performance speedup of all workloads (direct-mapped)

queue size for both set-associative and direct-mapped caches, the relatively higher pressure on the read queue in the set-associative design will decrease the chance of buffering the LRs, which in turn causes higher interference and results in lower performance.

With Remapping. We also conduct a second set of experiments, where we incorporate the permutation-based remapping (XOR) scheme [9] in all the designs. This scheme was originally proposed to reduce the number of RWI triggered row buffer conflicts for DRAM memory. Fig. 9 shows the speedups of all the designs with remapping, normalized to CD without remapping. As we can see from the figure, the addition of remapping significantly improves the performance for both direct-mapped and set-associative organizations. This is because remapping reduces the number of read-read-conflicts by remapping tag accesses to different banks (similar to reducing RWC in DRAM memory). Among designs with remapping, ROD has the worst performance. This is because ROD already reduces the number of RRCs and hence does not benefit much from remapping – but continues to suffer from the penalty of turnaround overheads. The overall performance of CD on the other hand is improved by 22.1% and 16.2% for direct-mapped and set-associative caches respectively. Again, DCA provides the maximum performance improvement of 29% and 23.7% in direct-mapped and set-associative caches respectively. It is worth noting that DCA (with remapping) improves upon CD (with remapping) by 7.5% and 7.0% for direct-mapped and set-associative caches respectively. This is because, in CD, the remapping scheme only helps in reducing row conflicts from RRC, but the problem of read priority inversion still persists. DCA on the other hand also reduces the impact of priority inversion by distinguishing between PRs and LRs.

Individual workloads. Figures 11 and 10 show speedups for individual workloads for direct-mapped and set-associative caches respectively. The speedups are normalized to CD (without remapping). We use *XOR+* prefix to represent designs with the remapping scheme. From these figures, we can see that the performance improvement trends seen in Fig. 8 and Fig. 9 are consistent across all benchmarks.

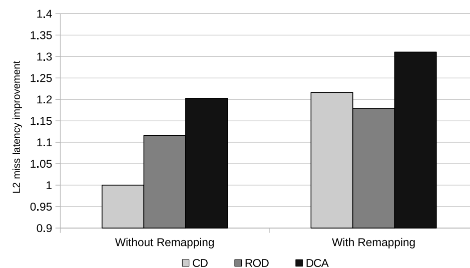

Fig. 12. L2 miss latency improvement — set-associative

B. *Miss Latency Improvement*

Set-associative. Fig. 12 shows the L2 miss latency improvement of various designs across all workloads for a set-associative cache, where results are normalized to CD. Without the remapping scheme [9], as the figure shows, CD has the worst miss latency amongst all designs. DCA improves the miss latency by 20% over the CD, while ROD improves it by 11%. With remapping, ROD becomes the worst performing among all designs. DCA has an improvement of 31%, whereas CD and ROD improve by 21.2% and 17.9% respectively. Thus, DCA (with remapping) improves upon CD (with remapping) by 9.8%.

Direct-mapped. For the direct-mapped cache, as we can see from Fig. 13, CD is still the worst in terms of per-

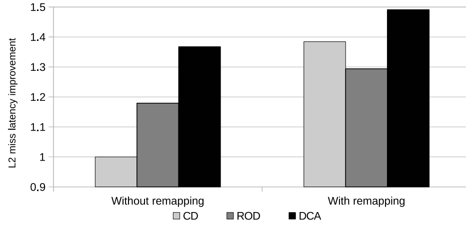

Fig. 13. L2 miss latency improvement — direct-mapped setting

formance amongst all designs without remapping. Similar to set-associative cache, in direct-mapped setting, ROD has an average improvement of 20% and DCA has an average improvement of 40%. With the remapping scheme, ROD has the worst performance among all designs. DCA has an improvement of 52%, whereas CD and ROD improve by 40% and 31% respectively. Thus, DCA (with remapping) improves upon CD (with remapping) by 12%.

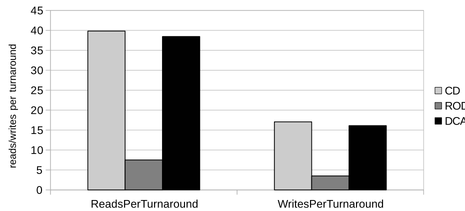

Fig. 14. Accesses per turnaround (the higher the better) — set-associative

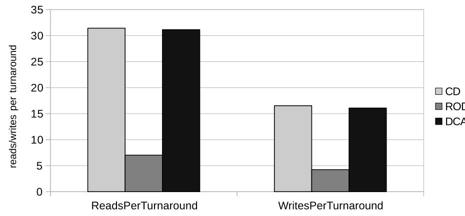

Fig. 15. Accesses per turnaround (the higher the better) — direct-mapped

C. Turnarounds

As we discussed in §II-A, a DRAM bus can only operate in either read or write mode at a given point of time. One of the design principles of DCA is to minimize the overhead of turnarounds. In this experiment, we analyze the number of accesses per turnaround for all designs. We observe that the remapping scheme has no impact on the number of turnarounds and hence we only show the results without remapping scheme. Fig.14 (set-associative) and Fig.15 (direct-mapped) show the number of read/write accesses per turnaround for all designs. Because CD places all the read accesses (irrespective of the request type) in the read queue, the number of read/write accesses that it processes per turnaround is the highest among all designs. On the other hand, ROD is only able to process about a third of read/write accesses per turnaround, as compared to CD. Finally, DCA is able to process almost the same number of read/write accesses as CD per turnaround. This explains why the performance of DCA is better than ROD (Fig. 8).

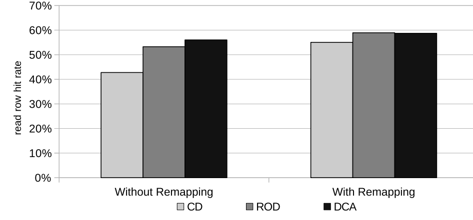

Fig. 16. Row buffer hit rate — set-associative

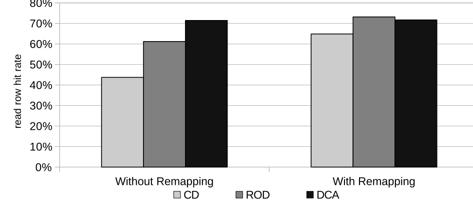

Fig. 17. Row buffer hit rate — direct-mapped

D. Row Buffer Hit Rate

In this study, we show the row buffer hit rate for read accesses. As we can see from Fig. 16, DCA has better row buffer hit rate than CD. This is because of two reasons. First, DCA avoids read-read-conflicts even without remapping. Second, buffering and passive scheduling of LRs allows the DCA scheduler to optimally schedule LRs and maximize row buffer locality. DCA achieves a row buffer hit rate of about 70% with/without the remapping scheme in direct-mapped and close to 60% in set-associative cache. It is worth noting that even though ROD with the remapping scheme has a slightly higher row buffer hit rate than DCA, it suffers from high turnaround overhead as we discussed in the previous section. That is why DCA outperforms ROD with/without the remapping scheme.

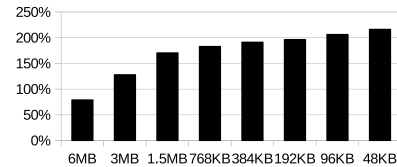

Fig. 18. Tag accesses for various tag cache sizes normalized to no tag cache

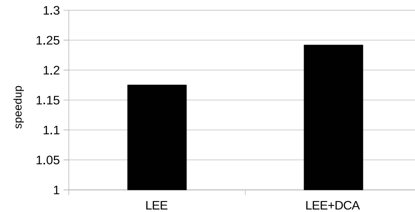

Fig. 19. Performance speedup under DRAM-aware writebacks scheme [20]

VII. Related Work

A number of recent works on die-stacked DRAMs have shown that this technology is a promising step in the direction of bridging the gap between on-chip caches and off-chip

memory. While some of them have proposed using this die-stack DRAM as part of main memory [21], [22], [23], [24], others have proposed to use it as a cache [1], [3], [4], [5], [6], [7], [14], [25].

DRAM caches allow for workloads with working set sizes on the order of hundreds of megabytes or even a few gigabytes to be cached. However, engineering a DRAM cache is not without its challenges. In particular, where to store DRAM cache tags is a non-trivial question that has attracted significant attention. Some prior studies [1], [25] have proposed to store cache tags in the SRAM. By storing tags in SRAM, it can reduce both access latency and miss penalty. However, these works need to use a page-based block (about 2KB) to reduce the space requirement in costly SRAM. Regardless, this design does not scale to large DRAM cache sizes [5]. Therefore, many recent works [3], [4], [5], [6], [7], [8], [14] have proposed to store these cache tags in the die-stacked DRAM array itself. Among them, Loh and Hill [6] were the first to establish the tags-in-DRAM design and propose a MissMap to skip accessing the DRAM cache on a miss. Qureshi et al. [7] and Sim et al. [14] also propose miss predictors that avoid the penalty of a DRAM cache miss with a smaller area overhead compared to the MissMap. Taking an alternative approach to using a dedicated storage for tags, recent works [26], [27] have proposed a tagless DRAM cache, where they use a cache-map TLB (c-TLB) to provide virtual-to-cache address mapping. While these works provide a promising alternative, the design is fundamentally coupled with the memory-management-unit and therefore requires OS modifications.

In this paper, we target a different problem pertaining to DRAM caches, one that has not been explored previously. We address the problem of how to effectively schedule DRAM cache's requests. Although our work is based on the set-up used by Loh's [6] and Qureshi's [7] organizations, our work is orthogonal and can be applied to most of existing tags-in-DRAM designs. In addition, for DRAM caches with separate tag storage, DCA can still be used for solving the priority inversion problem.

A number of recent works [3], [4], [28] have proposed caching/prefetching the tags of the DRAM cache in an SRAM buffer (with the tags backed up in the DRAM cache), in order to reduce the latency of accessing the tags. With a tag cache, however, the number of tag accesses to the DRAM does not reduce. In fact, our study based on the tag cache design by Huang et al. [4] shows that the number of tag accesses to the DRAM cache could even increase. As shown in Fig. 18, for a 256 MB (12 MB of tags) DRAM cache, the number of DRAM tag accesses is doubled even for a 192 KB tag cache. The reason for this is because tag accesses in a tag cache do not show much temporal locality, and most of the benefit comes from prefetching nearby tags due to spatial locality [4], [8]. This is not surprising: because the tag cache's size is smaller than the tag size of the last-level shared cache (around 384KB for a 8MB cache), it is unlikely to gain much from temporal locality. Therefore, adding a tag cache will only exacerbate the DRAM cache scheduling problem.

In BEAR [8], Chou et. al introduce techniques to reduce the number of DRAM cache accesses including a *Bandwidth Efficient Writeback Probe* which can remove read tag accesses for writeback hits. While effective, BEAR specifically targets an inclusive direct-mapped DRAM cache design. On the other hand, there have been a number of proposals that have advocated a set-associative design [3], [4] and also non-inclusive DRAM caches [29]. Regardless, our work is orthogonal and applicable for effectively scheduling the residual accesses in BEAR.

Several prior works [9], [20], [30] target the problem of writeback-caused row conflicts in the DRAM memory. Zhang et al. proposed a permutation-based remapping scheme for mitigating the RWC issue. Based on Rau's work [31], the permutation-based scheme generates a new *addressing* address by XORing the *original bank index* with *k-bit* (first N-bit of page index). The remapping scheme is effective in mitigating RRC problem in DRAM cache, as evidenced from our experiments in §VI-A (with remapping) – however, DCA can additionally mitigate read priority inversion.

More recently, Stuecheli et al. proposed a *virtual write queue* [30] which combines the memory controller's write queue with the last-level cache's *LRU way(s)* to form a virtual queue. This allows the scheduler algorithm to schedule the writeback before it actually appears in the write queue. In addition, Lee et al. propose a last-level cache writeback policy [20] to avoid reads and writes from being issued to DRAM at the same time. However, none of the above works address the problems associated with tag accesses and cache refills which are unique to a DRAM cache and hence, our work is orthogonal to the above works. This is evidenced by our study in which we include Lee's RWC scheme in the upper level cache (which is L2 in our setup). As shown in Fig. 19, we observe that LEE+RWC can continue to outperform LEE scheme by 7% in a direct-mapped cache.

A number of prior works have proposed application-aware scheduling policies that are designed for reducing application interference in a parallel environment [32], [33], [34], [11]. While our work uses BLISS as our scheduling algorithm, our proposal is orthogonal and can take advantage of the above scheduling algorithms. In addition, Zhao et al [35] propose a DRAM control scheme optimized for log-based workloads for systems with persistent memory. Zhao et al's work and our work share the similar theme which is specializing memory controller for a particular requirement. Whereas we propose a DRAM controller targeted towards DRAM caches, they tailor the memory controller for persistent memory.

VIII. Conclusion

Recent studies have proposed DRAM cache designs that maintain tags in the DRAM cache, which increases the complexity of a DRAM cache access. In this paper, we have addressed the problem of how to effectively schedule these DRAM cache accesses. A conventional DRAM controller classifies accesses into only two categories: reads and writes. This simple two way classification is not suitable for a DRAM

cache because different read accesses can have different priorities depending on the requests they correspond to, and depending on whether they are data or tag reads.

In this work, we have studied potential designs based on conventional DRAM controller and analyzed their limitations. We have proposed a DRAM-Cache-Aware (DCA) DRAM controller that is aware of DRAM-cache-specific tag accesses, and prioritizes tag accesses in the read queue based on their request type. We have also proposed an opportunistic flushing scheme (OFS) for flushing low-priority tag reads from the read queue.

Our experiments have showed that DCA improves upon CD (a naive design derived from a conventional DRAM controller) by 16.4% for a 15-way set-associative DRAM cache. For a direct-mapped cache design, which allows tag and data to be read in one single DRAM burst, the corresponding improvement over CD is 20.8%.

IX. Acknowledgements

We would like to thank Boris Grot and the anonymous reviewers for their helpful comments. This work is supported by EPSRC grants EP/M001202/1 and EP/M027317/1 to the University of Edinburgh.

References

[1] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian, "Chop: Adaptive filter-based dram caching for cmp server platforms," in *International Conference on High-Performance Computer Architecture, HPCA*, 2010.

[2] L. Zhao, R. R. Iyer, R. Illikkal, and D. Newell, "Exploring dram cache architectures for cmp server platforms," in *ICCD*, 2007.

[3] N. Gulur, G. R., R. Manikantan, and M. Mehendale, "Bi-modal dram cache: Improving hit rate, hit latency and bandwidth," in *IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2014.

[4] C. Huang and V. Nagarajan, "ATCache: reducing DRAM cache latency via a small SRAM tag cache," in *International Conference on Parallel Architectures and Compilation, PACT*, 2014.

[5] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi, "Unison cache: A scalable and effective die-stacked dram cache," in *IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2014.

[6] G. H. Loh and M. D. Hill, "Efficiently enabling conventional block sizes for very large die-stacked dram caches," in *IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2011.

[7] M. K. Qureshi and G. H. Loh, "Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design," in *IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2012.

[8] C. Chou, A. Jaleel, and M. K. Qureshi, "BEAR: techniques for mitigating bandwidth bloat in gigascale DRAM caches," in *International Symposium on Computer Architecture, ISCA*, 2015.

[9] Z. Zhang, Z. Zhu, and X. Zhang, "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality," in *IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2000.

[10] B. Jacob, S. Ng, and D. Wang, *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[11] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, "The blacklisting memory scheduler: Balancing performance, fairness and complexity," *CoRR*, vol. abs/1504.00390, 2015.

[12] A. Jaleel, K. B. Theobald, S. C. S. Jr., and J. S. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," in *International Symposium on Computer Architecture, ISCA*, 2010.

[13] N. L. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. G. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, 2011.

[14] J. Sim, G. H. Loh, H. Kim, M. O'Connor, and M. Thottethodi, "A mostly-clean dram cache for effective hit speculation and self-balancing dispatch," in *IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2012.

[15] S. Eyerman and L. Eeckhout, "Restating the case for weighted-ipc metrics to evaluate multiprogram workload performance," *Computer Architecture Letters*, vol. 13, no. 2, 2014.

[16] G. H. Loh and M. D. Hill, "Addendum of supporting very large dram caches with compound access scheduling and missmaps," 2012.

[17] JEDEC. (2012) Ddr3 sdram standard. [Online]. Available: http://www.jedec.org/standards-documents/docs/jesd-79-3d

[18] G. H. Loh, "Extending the effectiveness of 3d-stacked dram caches with an adaptive multi-queue policy," in *IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2009.

[19] JEDEC. (2014) Wide i/o 2 (wideio2). [Online]. Available: http://www.jedec.org/standards-documents/results/jesd229

[20] 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, The University of Texas at Austin, TR-HPS-2010-002*, 2010.

[21] C. Chou, A. Jaleel, and M. K. Qureshi, "Cameo:a two-level memory organization with capacity of main memory and flexibility of hardware-managed cache," in *IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2014.

[22] C. C. Liu, I. Ganusov, M. Burtscher, and S. Tiwari, "Bridging the processor-memory performance gapwith 3d IC technology," *IEEE Design & Test of Computers*, vol. 22, no. 6, 2005.

[23] J. Sim, A. R. Alameldeen, Z. Chishti, C. Wilkerson, and H. Kim, "Transparent hardware management of stacked dram as part of memory," in *IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2014.

[24] D. H. Woo, N. H. Seong, D. L. Lewis, and H.-H. S. Lee, "An optimized 3d-stacked memory architecture by exploiting excessive, high-density tsv bandwidth," in *International Conference on High-Performance Computer Architecture, HPCA*, 2010.

[25] D. Jevdjic, S. Volos, and B. Falsafi, "Die-stacked DRAM caches for servers: hit ratio, latency, or bandwidth? have it all with footprint cache," in *International Symposium on Computer Architecture, ISCA*, 2013. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485957

[26] Y. Lee, J. Kim, H. Jang, H. Yang, J. Kim, J. Jeong, and J. W. Lee, "A fully associative, tagless DRAM cache," in *International Symposium on Computer Architecture, ISCA*, 2015.

[27] H. Jang, Y. Lee, J. Kim, Y. Kim, J. Kim, J. Jeong, and J. W. Lee, "A fully associative, tagless DRAM cache," in *International Conference on High-Performance Computer Architecture, HPCA*, 2016.

[28] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan, "Enabling efficient and scalable hybrid memories using fine-granularity DRAM cache management," *Computer Architecture Letters*, vol. 11, no. 2, 2012.

[29] C.-C. Huang, R. Kumar, M. Elver, B. Grot, and V. Nagarajan, "C3D: mitigating the numa bottleneck via coherent dram caches," in *Annual IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2016.

[30] J. Stuecheli, D. Kaseridis, D. Daly, H. C. Hunter, and L. John, "Coordinating DRAM and last-level-cache policies with the virtual write queue," *IEEE Micro*, vol. 31, no. 1, 2011.

[31] B. R. Rau, "Pseudo-randomly interleaved memory," in *International Symposium on Computer Architecture, ISCA*, 1991.

[32] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems," in *International Symposium on Computer Architecture, ISCA*, 2008.

[33] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *International Conference on High-Performance Computer Architecture, HPCA*, 2010.

[34] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt, "Parallel application memory scheduling," in *IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2011.

[35] J. Zhao, O. Mutlu, and Y. Xie, "FIRM: fair and high-performance memory control for persistent memory systems," in *IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2014.