Row-Buffer Hit Harvesting in Orchestrated Last-Level Cache and DRAM Scheduling for Heterogeneous Multicore Systems

Yang Song[†], Olivier Alavoine[‡] and Bill Lin[†]
[†]Electrical and Computer Engineering Department, University of California San Diego, La Jolla, CA, USA
[‡]Qualcomm Inc., San Diego, CA, USA
y6song@eng.ucsd.edu

Abstract*—In heterogeneous multicore systems, the memory subsystem, including the last-level cache and DRAM, is widely shared among the CPU, the GPU, and the real-time cores. Due to their distinct memory traffic patterns, heterogeneous cores result in more frequent cache misses at the last-level cache. As cache misses travel through the memory subsystem, two schedulers are involved for the last-level cache and DRAM respectively. Prior studies treated the scheduling of the last-level cache and DRAM as independent stages. However, with no orchestration and limited visibility of memory traffic, neither scheduling stage is able to ensure optimal scheduling decisions for memory efficiency. Unnecessary precharges and row activations happen in DRAM when the memory scheduler is ignorant of incoming cache misses and DRAM row-buffer states are invisible to the last-level cache. In this paper, we propose a unified memory controller for the the last-level cache and DRAM with orchestrated schedulers. The memory scheduler harvests row-buffer hit opportunities in cache request buffers during spare time without inducing significant implementation cost. Extensive evaluations show that the proposed controller improves the total memory bandwidth of DRAM by 16.8% on average and saves DRAM energy by up to 29.7% while achieving comparable CPU IPC. In addition, we explore the impact of last-level cache bypassing techniques on the proposed memory controller.

***Index Terms*—memory subsystem, row-buffer hit, memory efficiency, heterogeneous MPSoCs**

I. Introduction

Modern heterogeneous multicore SoC architectures [1], [2] have been widely deployed in mobile devices thanks to their energy efficiency. These multicore SoCs typically integrate a diverse collection of cores. Besides general-purpose cores like the CPU for running applications, most heterogeneous cores are dedicated to specific functions, including the GPU, multimedia processing units such as video decoders and camera, and system cores such as the GPS and USB units, and so on. To save cost and energy, heterogeneous cores commonly share resources, among which, the sharing of the memory subsystem is the most challenging because memory performance often has a direct and substantial impact on the system performance.

The traditional architecture of a shared memory subsystem for heterogeneous multicore systems is shown in Fig. 1. The last-level cache, as the first merging point of heterogeneous memory traffic, arbitrates among requests from different cores and handles memory coherence to ensure the correctness of shared data accesses. Cache misses and writebacks from the last-level cache are forwarded to the memory controller whereupon these requests are buffered in the transaction queue and awaiting to be scheduled to the DRAM. At each clock cycle, up to one memory request can be selected by the memory scheduler based on certain scheduling policy. Once a scheduling decision has been made, the selected request is sent to the command generator designated to the destination memory bank. The command generator generates commands to operate the DRAM following the DRAM memory access protocol [3]. In a DRAM memory channel, data storage is organized into multiple hierarchies including (from top to bottom) ranks, banks, rows and columns. While DRAM ranks and banks can be accessed in parallel, only one row can be active in each bank. To read or write a column in an inactive row, the bank must be precharged (any active row becomes then closed). Then the bank loads the target row into the row-buffer (i.e. row activation operation) before performing the actual column read or write access.

An inherent limitation of DRAM memory is that precharge and row activation operations introduce extra power and time penalties without contributing to actual data accesses. Hence the memory scheduler needs to take into account the row-buffer states in each bank to circumvent unnecessary precharges and activations and improve memory efficiency. First-Ready First-Come-First-Serve (FR-FCFS) policy [4] is prevalently used to prioritize requests to open rows when available or older requests otherwise. The minimalist scheduling policy [5] was proposed to garner open row-buffer hits between precharges and activations without causing unfairness among different cores/threads. Batching mechanism [6] has also been used to collect row-buffer hits and relieve the memory interference against latency-sensitive requests. Although research on last-level cache design in multicore systems [7]–[9] usually deals with cache partitioning issues, discussions on optimizing cache traffic for memory efficiency have also drawn attentions in recent years. Several cache replacement and eviction policies have been proposed [10]–[12] in consideration of memory efficiency in terms of DRAM power and average memory latency.

The last-level cache controller and the memory controller are usually seen as separate units that make independent scheduling decisions. However, as incoming cache requests are invisible to the memory scheduler and row-buffer states are invisible to the cache controller, limited memory visibility on each side often leads to sub-optimal scheduling decisions. These uncoordinated decisions may eventually suppress the memory efficiency when either controller ignores the chances

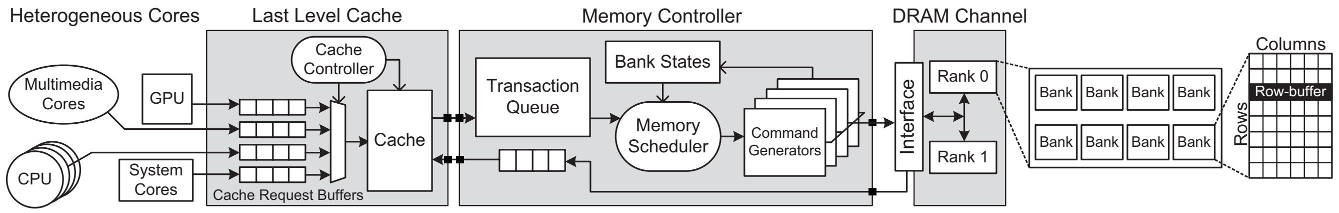


Fig. 1. Traditional architecture of the memory subsystem including the last-level cache, the memory controller and a DRAM memory channel for heterogeneous multicore systems. Private caches and on-chip interconnects are omitted.

for row-buffer hits on the other side. In addition, the increased cache miss rate by heterogeneous traffic makes the problem even worse by bringing more traffic to DRAM from the last-level cache.

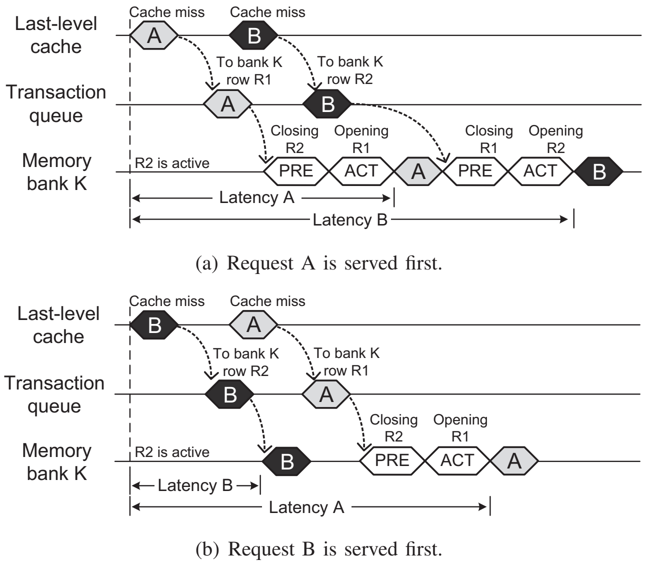
In this work, we propose a unified memory controller for the last-level cache and DRAM. The goal of the unified memory controller is to endow more visibility to memory scheduling and orchestrate scheduling decisions for the last-level cache and DRAM. The contributions of our work can be summarized as follows.

- We identify the limitations of separate scheduling stages for the last-level cache and DRAM. With limited visibility on both sides, the last-level cache controller and the memory controller are not able to make the best decisions to improve memory efficiency.
- We propose the design of a unified memory controller with orchestrated schedulers for the last-level cache and DRAM. Given access to cache request buffers, the memory scheduler gains more visibility and harvests more fleeting row-buffer hit opportunities, without raising hardware complexity.
- We evaluate the proposed design using memory traffic of a next-generation heterogeneous multicore SoC and show that row-buffer hits and DRAM bandwidth are improved by 22% and 16.8% respectively on average by the unified controller. DRAM energy can be saved by up to 29.7% with comparable CPU IPC achieved. We also explore the impact of cache bypassing on the unified controller.

The rest of this paper is organized as follows: Section II describes the background and motivation of our work. Section III presents the architecture of the proposed unified memory controller. The evaluation results and conclusions follow in Sections IV and V.

II. Background and Motivation

Architectures of the last-level cache and the memory controller have been widely discussed in recent years. Most prior work tackle these two units as independent memory fabric [4]–[9]. In contrast, fewer work have explored the interaction between the last-level cache and the memory controller. Stuecheli et al. [10] were the first to coordinate memory scheduling with cache writeback traffic. Their *virtual write queue* is implemented by LRU cache lines in the last-level cache, as an extension to the transaction queue in the memory controller. The virtual write queue improves the scheduler's visibility into memory locality and thus leads to better memory scheduling efficiency. Kim et al. [11] proposed a new scheduling scheme for the last-level cache in consideration of DRAM rank power states. The objective is to reduce DRAM power state transitions induced by evicted cache lines. Jeon et al. [12] extended the virtual write queue to include read requests when clustering memory requests and cache writebacks for open row-buffers. A comprehensive exploration of DRAM-aware eviction policies was also provided by the authors.



(a) Request A is served first.

(b) Request B is served first.

Fig. 2. An example of cache access ordering affecting memory efficiency. Destination of request A: bank K row $R1$. Destination of request B: bank K row $R2$. Initial active row in bank K: row $R2$.

Aforementioned work focus on the DRAM-aware management of last-level cache writebacks. However, cache misses can also affect the efficiency of memory scheduling. Fig. 2 illustrates an example in which cache access ordering determines the number of row activations and precharges in DRAM. Assume cache requests A and B are both waiting at the last-level cache in the beginning. The target addresses of requests A and B are located in the same memory bank (bank K) but different rows ($R1$ and $R2$ respectively). The target row by request B, row $R2$, is initially active. In the case of cache misses for both requests, if request A is served first by the last-level cache (in Fig. 2(a)), it will arrive at the transaction queue earlier than request B. Without knowing the existence of request B, the memory scheduler precharges (i.e. closes) row $R2$ and activates (i.e. opens) row $R1$ for request A. When request B arrives later, the scheduler will have to close row $R1$ and re-open $R2$. In the other scenario (in Fig. 2(b)), request B accesses the last-level cache first and gets to be scheduled to row $R2$ before this row is closed for request A, avoiding extra precharge and activation operations.

By comparison, memory scheduling is less efficient in the first scenario because the unnecessary precharge and activation operations result in higher DRAM power and average memory latency. Yet, this can be prevented if row-buffer states are visible to the last-level cache or incoming cache requests are visible to the memory scheduler so that orchestrated scheduling decisions can be made to ensure request B arrives before row $R2$ is closed.

What's worse, the overall miss rate of the last-level cache is always higher when it is being shared by heterogeneous cores including real-time multimedia processing units and the GPU [7], [9]. Traffic from such heterogeneous cores shows low temporal locality, but travels through the last-level cache so that memory coherence can be managed (since these cores share the same address space with the CPU). In addition, due to their high thread-level parallelism, some heterogeneous cores (e.g. the GPU and video decoder) have high cache access rates, which further raises the frequency of cache misses. Therefore, the problem depicted in Fig. 2 is fairly common in heterogeneous multicore systems. Although last-level cache bypassing [8] by real-time traffic has been proposed to avoid polluting the last-level cache with non-reusable data and reduce cache misses, not all real-time traffic can bypass the last-level cache.

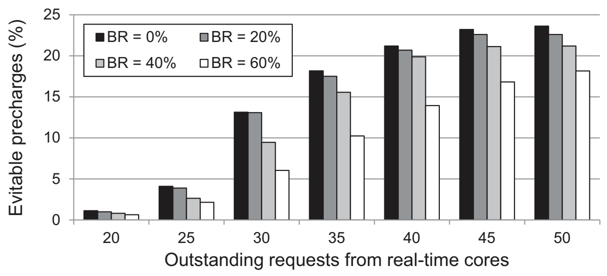


Fig. 3. Percentage of evitable precharges vs. the number of outstanding real-time requests and cache bypassing ratio (BR) for real-time traffic. Scheduling decisions for the last-level cache and DRAM are independent from each other. CPU test case 4 and real-time application *GPU_shader* are used for simulation (Tab. II). Experiment setup is discussed in Section IV.

We identify a precharge operation as *evitable* when there are last-level cache requests which target at the row to be precharged and will turn out to be cache misses later on. For example, the precharge operation for $R2$ in Fig. 2(a) is evitable because a potential row-buffer hit by request B is waiting at the last-level cache.

The number of evitable precharges indicates the potential for memory efficiency improvement if more visibility is given to the scheduler. Fig. 3 shows the percentage of precharge operations that are evitable as the number of outstanding real-time requests and real-time cache bypassing ratio[1] vary. As can be seen, more evitable precharges happen when more outstanding real-time requests are injected. Without cache bypassing (BR = 0%), the percentage of evitable precharges saturates around 23% after the number of outstanding real-time requests reaches 50, which is not too much considering that there are plenty of heterogeneous cores, including memory-intense ones such as the GPU. Even when 60% of real-time traffic bypasses the last-level cache, the percentage of evitable precharges can still reach up to 18% given enough outstanding requests.

[1]Cache bypassing ratio is computed as the amount of cache bypassing traffic divided by the total amount of memory traffic.

III. Unified Memory Controller with Orchestrated Schedulers

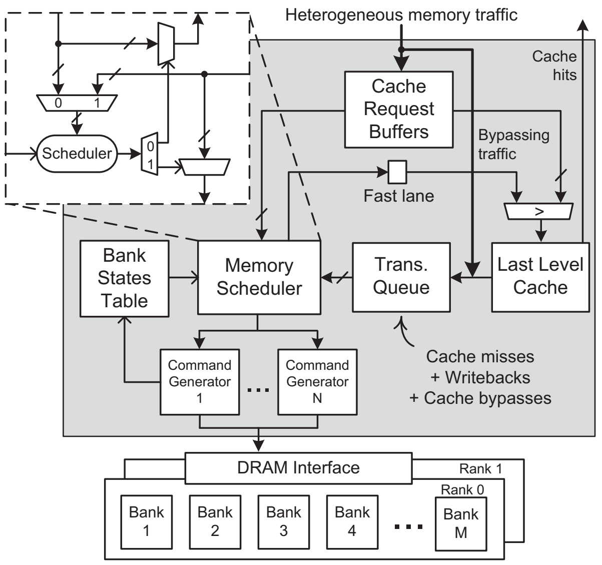


Fig. 4. Architecture of the unified controller for the last-level cache and DRAM.

In this section, we present the unified memory controller with improved memory visibility and orchestrated scheduling for the last-level cache and DRAM. Architecture of the unified controller is shown in Fig. 4. Similar to the conventional design (Fig. 1), most memory traffic, except cache bypassing traffic, still travels through cache request buffers, the last-level cache, the transaction queue and DRAM. Yet the memory scheduler now has access to cache request buffers so that row-buffer hit opportunities can be detected and harvested as early as the cache requests arrive in the memory subsystem. A short-cut for cache requests, i.e. the *fast lane*, is introduced to expedite the delivery of potential row-buffer hits. More details follow in the rest of this section.

A. Memory Scheduling and Row-buffer Hit Harvesting

In the unified memory controller, the memory scheduler works in two modes: memory scheduling and row-buffer hit harvesting.

In the scheduling mode, the scheduler performs the conventional memory scheduling task. Specifically, the scheduler arbitrates among requests in the transaction queue including cache misses and cache writebacks from the last-level cache, and cache bypassing memory requests. This process requires information on the states of DRAM banks and row-buffers, which is maintained on the bank states table. To avoid unnecessary precharges and row activations, preferentially a request for an idle bank and an active row is chosen. The chosen request will be forwarded to one of the per-bank command generators. The command generator will convert the selected request into one or multiple commands to operate

TABLE I
SIMULATION PARAMETERS.

Number of Outstanding Requests			
CPU cores	16	Heterogeneous cores	50
Last-level Cache			
Size	2MB	Associativity	8
Miss latency	4 cycles	Hit latency	4 cycles
Request buffers	40		
Memory Controller			
Read queue size	30	Write queue size	30
DRAM			
Volume	4GB	Max I/O bus freq.	1866MHz
CL-tRCD-tRP (cycles)	36-34-34	tWTR-tRTP-tWR (cycles)	19-14-34
tRRD-tFAW (cycles)	19-75	Channels-Ranks-Banks	1-2-8

TABLE II
BENCHMARK DESCRIPTIONS.

CPU	Description
case 1	art–mcf–sjeng–sphinx3
case 2	art–mcf–sjeng–hmmer
case 3	mcf–sjeng–sphinx3–hmmer
case 4	art–mcf–sphinx3–hmmer
Real-time	**Description**
camcorder a	Camcorder application in UHD format at 60FPS with snapshot (one picture per second)
camcorder b	Camcorder application using dual cameras at 30FPS
display_panel	Dump display panel refreshing in WQXGA resolution at 60FPS (RGB888 pixel format)
GPU_shader	GPU shader composition of 4 ARGB layers

the DRAM bank, according to the low-level DRAM timing constraints. Note that even though the memory scheduler is capable of performing scheduling at every cycle, it takes time for DRAM to execute each operation command. Besides, there are constraints on how often one can activate row-buffers in order to limit dynamic power [3]. Therefore, most of the time the scheduler is waiting to issue new operations. We utilize these idle cycles by introducing the row-buffer hit harvesting mode to the scheduler.

In the harvesting mode, the scheduler scans through extant read requests in cache request buffers. If the target address of a read request is located in an open row or a row to be opened within t_{miss} cycles, the scheduler will remove this request from the request buffer and send it to the last-level cache through the fast lane, Here, t_{miss} is the cache miss latency of the last-level cache. If more than one such requests are found, the oldest one will be selected. If no such requests can be found or the fast lane is busy, no cache requests will be harvested by the scheduler. To minimize the transmission latency on the fast lane, only one request buffer is used. Thus the request is visible to the last-level cache as soon as it is harvested by the scheduler. A pipelined last-level cache can receive a new request at each clock cycle. That guarantees that the request, if turns out to be a cache miss, will arrive at the transaction queue t_{miss} cycles after being harvested.

The purpose of having the harvesting mode is to capture cache requests that potentially will result in row-buffer hits and reduce their cache access latency so that they can finish cache lookup before the open rows are closed. Moreover, to avoid limiting memory throughput we do not hold the row-buffer open for the cache request on the fast lane in case it will end up as a cache hit.

B. Orchestrated Cache Request Scheduling

As mentioned above, the last-level cache fetches cache requests from the scheduler through the fast lane. Yet, since row-buffer hit harvesting is not being performed all the time, the scheduler is not able to deliver a cache request at each clock cycle. Thus the last-level cache still needs a local scheduler to arbitrate among requests from cache request buffers in case the fast lane is empty.

At the last-level cache, orchestration with the scheduler is done by prioritizing the requests arriving from the fast lane. One such scheduling policy is to always look up the cache request from the fast lane first, otherwise find one from cache request buffers based on the local policy. In our evaluation, we set the local scheduling policy as prioritizing read requests over writes, and old over new. QoS-aware policies can also be applied at the local scheduler with consideration of row-buffer hit opportunities from the fast lane.

C. Hardware Implementation and Cost

In the implementation, memory scheduling and row-buffer hit harvesting are implemented on the same scheduler (as in Fig. 4) in order to reutilize the arbitration logic. For both arbitrations, the logic contains a series of magnitude comparators, with multiple inputs and a single output. Apart from slightly different policies, the major difference between these two arbitrations are input sources and output destinations. For memory scheduling, the arbiter receives memory requests from the transaction queue as the inputs and outputs the chosen request to a command generator. For row-buffer hit harvesting, the arbiter accepts existing cache read requests as the inputs and outputs the result to the fast lane.

Multiplexers are used to direct memory traffic inside the scheduler according to the working mode, as shown in Fig. 4 where indices 1 and 0 at multiplexers represent the scheduling and harvesting modes respectively. To switch between two working modes, the scheduler only needs to set the selection signals at the multiplexers for outputs and inputs. The arbitration result contains the index of the chosen request and is used to multiplex input requests from either the transaction queue or cache request buffers.

In addition, the fast lane contains one request buffer and adds one more input at the local scheduler for the last-level cache. Overall, the implementation of the orchestrated schedulers do not require complex logic with significant cost.

IV. EVALUATION

In this section, the proposed unified memory controller will be tested in comparison with the conventional design with independent schedulers. Performance metrics of the memory subsystem, including row-buffer hits, DRAM bandwidth and energy will be evaluated, as well as CPU IPC.

A. Methodology

The unified memory controller is implemented on basis of DRAMSim2 [13] using LPDDR4 timing model. We perform cycle-accurate simulations of the memory subsystem, including the last-level cache and DRAM. The simulation parameters are listed in Tab. I. On top of the simulator, a heterogeneous system is emulated with four CPU cores and real-time cores such as the GPU and the video decoder etc. Each CPU core has one application from SPEC 2000/2006 benchmarks running and generating traffic for the last-level cache. For real-time cores, we collect memory traffic per data stream generated by all real-time cores in a next-generation heterogeneous multi-core SoC [1] while running real life multimedia applications. To emulate cache bypassing for real-time traffic, we randomly pick real-time data streams whose requests will always bypass the last-level cache. To achieve a specific bypassing ratio (BR), say 20%, we pick 20% of data streams from real-time cores and route their memory traffic directly to the transaction queue. Detailed descriptions of CPU test cases and real-time applications are listed in Tab. II.

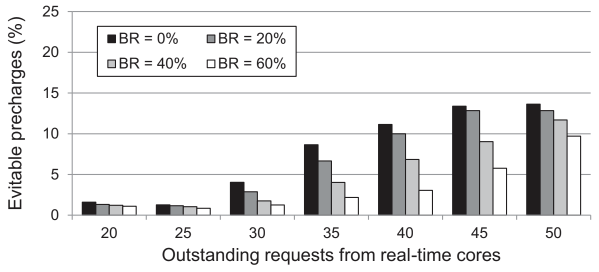


Fig. 5. Percentage of evitable precharges vs. the number of outstanding real-time requests and cache bypassing ratio (BR) for real-time traffic. The unified memory controller is deployed. Test case 4 and real-time application *GPU_shader* are used for the CPU and real-time cores (Tab. II).

B. Row-buffer Hits and DRAM Bandwidth Improvement

To begin with, we test the unified memory controller in terms of evitable precharges. The same CPU test case and real-time application as in Fig. 3 are applied. As shown in Fig. 5, the unified controller lowers the percentage of evitable precharges to below 14% when there is no cache bypassing traffic. As more real-time traffic bypasses the last-level cache, there are fewer row-buffer hit opportunities in cache request buffers. Therefore, the percentage of evitable precharges lowers as the bypassing ratio increases. When the ratio is 60%, about 10% precharges are evitable. In comparison with the conventional approach with separate scheduling stages (Fig. 3), the unified controller reduces the percentage of evitable precharges by about 10% in all cases.

By avoiding evitable precharges, the orchestrated schedulers improve the average number of row-buffer hits after each row activation. Fig. 6 shows the summary of normalized row-buffer hits by the unified controller in comparison with the conventional design with separate scheduling units. In each test case, the memory subsystem is simulated for one million cycles. To normalize row-buffer hit count, the average number of row-buffer hits by the unified controller is divided by that of the conventional controller. Without cache bypassing, the unified controller improves the number of row-buffer hits by 22% on average. Since last-level cache bypassing reduces the opportunities of row-buffer hit harvesting in cache request buffers, the improvement of row-buffer hits by the unified controller decreases as bypassing ratio increases. Yet, the average number of row-buffer hits by the unified controller can still be 17.6% higher than the conventional design, when 60% real-time traffic bypasses the last-level cache. Similar results can be observed in DRAM bandwidth. Fig. 7 shows the summary of normalized DRAM bandwidth by the unified controller. Without cache bypassing, the average improvement of DRAM bandwidth is 16.8%. The improvement shrinks to 13.9% as bypassing ratio is raised to 60%.

C. CPU IPC Comparison

On CPU IPC, the proposed controller with orchestrated schedulers has mixed impacts. The summary of CPU IPC by the unified controller, normalized with respect to the conventional controller, is shown in Fig. 8. Without cache bypassing, CPU IPC is suppressed by the unified controller as the average IPC drops 10%. This is because multimedia memory traffic from real-time cores often manifests higher spatial locality than CPU traffic. When row-buffer hit harvesting is performed, real-time traffic receives more service than CPU traffic which is slowed down as the result. However, when cache bypassing is applied, CPU traffic, faced with less competitions at the last-level cache, receives more benefits from row-buffer hit harvesting. Hence CPU IPC is gradually improved as bypassing ratio increases. When the ratio is 60%, the average CPU IPC by the unified controller ends up 10% higher than the conventional controller. Overall, performance of the unified controller and the conventional controller are comparable in terms of CPU IPC.

TABLE III
SUMMARY OF DRAM ENERGY SAVINGS BY THE UNIFIED CONTROLLER FOR PROCESSING 50000 MEMORY REQUESTS.

CPU	Real-time	Energy saving	Real-time	Energy saving
case 1	camcorder a	26.7%	camcorder b	29.7%
case 2		26.3%		29.2%
case 3		21.9%		24.8%
case 4		26.7%		29.7%
case 1	display_panel	22.5%	GPU_shader	21.1%
case 2		24.6%		20.4%
case 3		20.4%		13.1%
case 4		27.4%		20.8%

D. DRAM Energy Saving

Thanks to higher row-buffer hits and bandwidth, the unified controller helps DRAM finish processing the same number of memory requests with less time and power. We use an open source tool named DRAMPower [14] for command trace based power and energy estimation. Tab. III lists the total DRAM energy savings for all test cases by the unified memory controller compared with the baseline controller with independent schedulers. The proposed controller achieves lower DRAM energy in all cases. In the best case, the proposed controller saves DRAM energy by 29.7%.

V. CONCLUSIONS

In this paper, we show that existing controllers for the last-level cache and DRAM with independent schedulers often lead to unnecessary precharges and row activations in DRAM, due to their limited visibility into memory traffic. These evitable operations introduce extra delays and power, limiting memory efficiency. To improve the visibility for memory scheduling, we propose the unified memory controller with orchestrated schedulers for the last-level cache and DRAM. The unified

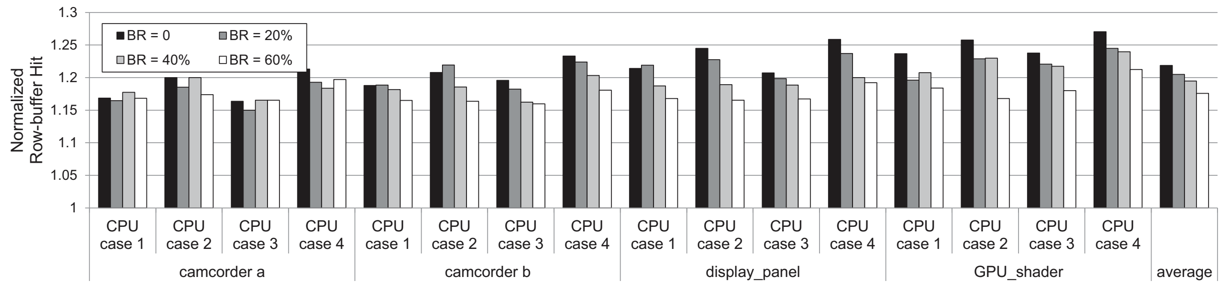


Fig. 6. Summary of *normalized row-buffer hits* in one million cycles by the proposed memory controller compared with the conventional design performing independent scheduling for the last-level cache and DRAM.

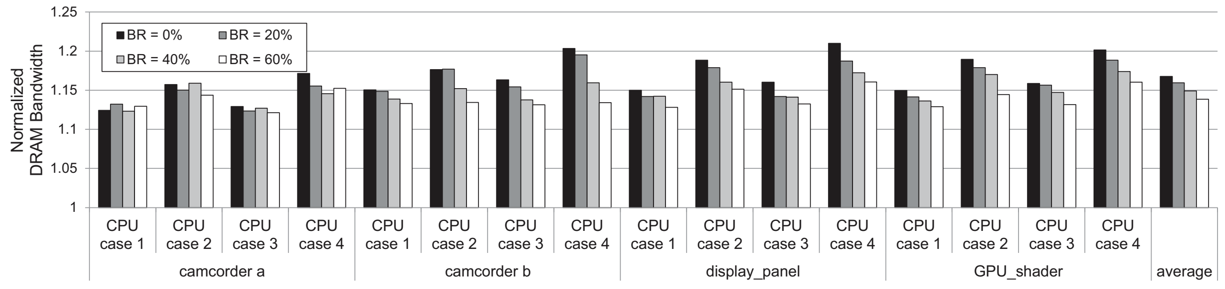


Fig. 7. Summary of *normalized DRAM bandwidth* in one million cycles by the proposed memory controller compared with the conventional design performing independent scheduling for the last-level cache and DRAM.

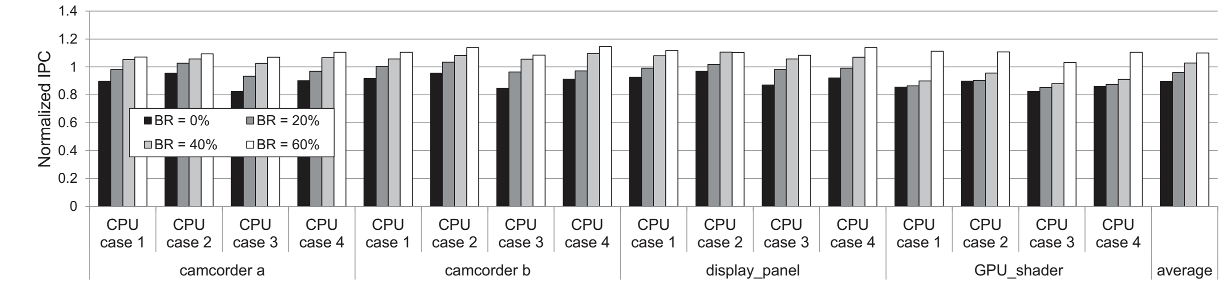


Fig. 8. Summary of *normalized CPU IPCs* in one million cycles by the proposed memory controller compared with the conventional design performing independent scheduling for the last-level cache and DRAM.

memory controller harvests more row-buffer hits and achieves higher DRAM bandwidth compared with the conventional design. According to our evaluations, the unified controller increases the number of row-buffer hits and DRAM bandwidth by 22% and 16.8% respectively on average. With up to 60% of real-time traffic bypassing the last-level cache, row-buffer hits and bandwidth can still be improved by 17.6% and 13.9% respectively on average. Further, the proposed unified controller saves DRAM energy by up to 29.7% while achieving comparable CPU IPC.

REFERENCES

[1] Qualcomm. Snapdragon 820. https://www.qualcomm.com/products/snapdragon/processors/820, 2015.

[2] NVIDIA. Tegra x1. http://www.nvidia.com/object/tegra-x1-processor.html, 2015.

[3] B. Jacob, S. Ng, and D. Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[4] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ACM ISCA*, 2000.

[5] D. Kaseridis, J. Stuecheli, and L. K. John. Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era. In *IEEE/ACM MICRO*, 2011.

[6] R. Ausavarungnirun, K. Chang, L. Subramanian, G. H. Loh, and O. Mutlu. Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems. In *ACM ISCA*, 2012.

[7] J. Lee and H. Kim. TAP: A TLP-aware cache management policy for a CPU-GPU heterogeneous architecture. In *ACM/IEEE HPCA*, 2012.

[8] V. Mekkat, A. Holey, P.-C. Yew, and A. Zhai. Managing shared last-level cache in a heterogeneous multicore processor. In *ACM/IEEE PACT*, 2013.

[9] P.-H. Wang, C.-H. Li, and C.-L. Yang. Latency sensitivity-based cache partitioning for heterogeneous multi-core architecture. In *ACM/IEEE DAC*, 2016.

[10] 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 *ACM ISCA*, 2010.

[11] S. Kim, S. Kim, and Y. Lee. DRAM power-aware rank scheduling. In *ACM/IEEE ISLPED*, 2012.

[12] M. Jeon, C. Li, A. L. Cox, and S. Rixner. Reducing DRAM row activations with eager read/write clustering. *ACM Trans. Archit. Code Optim.*, 10(4):43:1–43:25, Dec 2013.

[13] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob. DRAMSim: A memory-system simulator. In *SIGARCH Computer Architecture News*, 2005.

[14] K. Chandrasekar, C. Weis, Y. Li, S. Goossens, M. Jung, O. Naji, B. Akesson, N. Wehn, and K. Goossens. DRAMPower: Open-source DRAM power and energy estimation tool. http://www.drampower.info.