

The IPSI BgD Transactions on Advanced Research

Multi-, Inter-, and Trans-disciplinary Issues in Computer Science and Engineering

**A publication of IPSI Bgd Internet Research Society New York, Frankfurt, Tokyo, Belgrade
July 2018 Volume 14 Number 2 (ISSN 1820-4511)**

**Special issue: „Recent Advances in Overcoming Bottlenecks in Memory Systems
and Managing Memory Resources in GPU Systems“**

Guest Editors: Onur Mutlu, Saugata Ghose, and Rachata Ausavarungnirun

Table of Contents:

Guest Editor Introduction

Mutlu, Onur; Ghose, Saugata; and Ausavarungnirun, Rachata1

Predictable Performance and Fairness through Accurate Slowdown Estimation in Shared Main Memory Systems

Subramanian, Lavanya; Seshadri, Vivek; Kim, Yoongu; Jaiyen, Ben; and Mutlu, Onur5

High-Performance and Energy-Efficient Memory Scheduler Design for Heterogeneous Systems

Ausavarungnirun, Rachata; Loh, Gabriel H.; Subramanian, Lavanya; Chang, Kevin;
and Mutlu, Onur16

Exploiting DRAM Microarchitecture to Increase Memory-Level Parallelism

Kim, Yoongu; Seshadri, Vivek; Lee, Donghyuk; Liu, Jamie; and Mutlu, Onur28

Reducing DRAM Refresh Overheads with Refresh-Access Parallelism

Chang, Kevin K.; Lee, Donghyuk; Chishti, Zeshan; Alameldeen, Alaa R.; Wilkerson, Chris;
Kim, Yoongu; and Mutlu, Onur37

Exploiting Row-Level Temporal Locality in DRAM to Reduce the Memory Access Latency

Hassan, Hasan; Pekhimenko, Gennady; Vijaykumar, Nandita; Seshadri, Vivek;
Lee, Donghyuk; Ergin, Oguz; and Mutlu, Onur45

Heterogeneous-Reliability Memory: Exploiting Application-Level Memory Error Tolerance

Luo, Yixin; Govindan, Sriram; Sharma, Bikash; Santaniello, Mark; Meza, Justin;
Kansal, Aman; Liu, Jie; Khessib, Badriddine; Vaid, Kushagra; and Mutlu, Onur55

A Memory Controller with Row Buffer Locality Awareness for Hybrid Memory Systems Yoon, HanBin; Meza, Justin; Ausavarungnirun, Rachata; Harding, Rachael A.; and Mutlu, Onur	66
Decoupling GPU Programming Models from Resource Management for Enhanced Programming Ease, Portability, and Performance Vijaykumar, Nandita; Hsieh, Kevin; Pekhimenko, Gennady; Khan, Samira; Shrestha, Ashish; Ghose, Saugata; Jog, Adwait; Gibbons, Phillip B.; and Mutlu, Onur	76
Holistic Management of the GPGPU Memory Hierarchy to Manage Warp-level Latency Tolerance Ausavarungnirun, Rachata; Ghose, Saugata; Kayiran, Onur; Loh, Gabriel H.; Das, Chita R.; Kandemir, Mahmut T.; and Mutlu, Onur	86
Mosaic: An Application-Transparent Hardware–Software Cooperative Memory Manager for GPUs Ausavarungnirun, Rachata; Landgraf, Joshua; Miller, Vance; Ghose, Saugata; Gandhi, Jayneel; Rossbach, Christopher J.; and Mutlu, Onur	96

The IPSI BgD Internet Research Society

The Internet Research Society is an association of people with professional interest in the field of the Internet.
All members will receive these TRANSACTIONS upon payment of the annual Society membership fee of €500
(air mail printed matters delivery).

Member copies of Transactions are for personal use only
IPSI BGD TRANSACTIONS ON ADVANCED RESEARCH
www.internetjournals.net

STAFF		
Veljko Milutinovic, Co-Editor-in-Chief	Jakob Salom, Co-Editor-in-Chief	Nenad Korolija, Journal Manager
Department of Computer Engineering ETF University of Belgrade POB 35-54 Belgrade, Serbia Tel: (381) 64-1389281	Department of Computer Science Mathematical Institute of SANU University of Belgrade POB 367 Belgrade, Serbia Tel: (381) 64-8183030	Department of Computer Science IPSI Internet Research Society University of Belgrade POB 35-54 Belgrade, Serbia Tel: (381) 65-6725938
vm@eft.rs	jakob.salom@yahoo.com	nenadko@gmail.com
EDITORIAL BOARD		
Lipkovski, Aleksandar	Gonzalez, Victor	Milligan, Charles
The Faculty of Mathematics, Belgrade, Serbia	University of Oviedo, Gijon, Spain	Sun Microsystems, Colorado USA
Blaisten-Barojas, Estela	Janicic, Predrag	Kovacevic, Milos
George Mason University, Fairfax, Virginia USA	The Faculty of Mathematics, Belgrade Serbia	School of Civil Engineering, Belgrade Serbia
Crisp, Bob	Jutla, Dawn	Neuhold, Erich
University of Arkansas, Fayetteville, Arkansas USA	Sant Marry's University, Halifax Canada	Research Studios Austria, Vienna Austria
Domenici, Andrea	Karabeg, Dino	Piccardi, Massimo
University of Pisa, Pisa Italy	Oslo University, Oslo Norway	Sydney University of Technology, Sydney Australia
Flynn, Michael	Kiong, Tan Kok	Radenkovic, Bozidar
Stanford University, Palo Alto, California USA	National University of Singapore Singapore	Faculty of Organizational Sciences, Belgrade Serbia
Fujii, Hironori	Kovacevic, Branko	Rutledge, Chip
Fujii Labs, M.I.T., Tokyo Japan	School of Electrical Engineering, Belgrade Serbia	Purdue Discovery Park, Indiana USA
Ganascia, Jean-Luc	Patricelli, Frederic	Mester, Gyula
Paris University, Paris France	ICTEK Worldwide L'Aquila Italy	University of Szeged, Szeged Hungary

Guest Editor Introduction: Recent Advances in Overcoming Bottlenecks in Memory Systems and Managing Memory Resources in GPU Systems

Onur Mutlu^{1,2} Saugata Ghose² Rachata Ausavarungnirun²

¹*ETH Zürich* ²*Carnegie Mellon University*

Memory and storage systems are a fundamental system performance, energy, and reliability bottleneck in modern systems [5, 6, 7, 25, 28, 29]. This bottleneck is becoming increasingly severe due to (1) the very limited latency reductions in memory and storage devices over the last several years; (2) aggressive manufacturing process technology scaling and other techniques to improve memory density, such as multi-level cell technology, which increase the storage capacity of these devices, but introduce more raw bit errors and increase manufacturing process variation; (3) limited pin counts in chip packages, which prevent system designers from adding more and/or wider buses to increase bandwidth; (4) overwhelmingly data-intensive applications, which require high-bandwidth access to very large amounts of data; and (5) the increasing fraction of overall system energy consumed by memory systems and data movement. To make matters worse, it is becoming increasingly difficult to continue scaling these devices to smaller process technology nodes, and even though alternative emerging memory and storage technologies can potentially alleviate some of the shortcomings of existing memory and storage technologies, they also introduce *new* shortcomings that were previously absent. Therefore, there is a pressing need to comprehensively understand and mitigate these bottlenecks in both existing and emerging memory and storage systems and technologies.

This issue features extended summaries and retrospectives of some of the recent research done by our research group, SAFARI [33, 39], on (1) various critical problems in memory systems and (2) how memory system bottlenecks affect graphics processing unit (GPU) systems. As more applications share a single system, operations from each application can contend with each other at various shared components within the system. If left unmitigated, such contention can undermine many of the benefits of parallelism, by slowing down each application or thread of execution [24, 26, 27, 28, 29]. The compound effect of contention, high memory latency and access overheads, as well as inefficient management of resources, greatly degrades performance, quality-of-service (QoS), and energy efficiency. The ten works featured in this issue study several aspects of (1) inter-application interference in multicore systems, heterogeneous systems, and GPUs; (2) the growing overheads and expenses associated with growing memory densities and latencies; and (3) performance, programmability, and portability issues in modern GPUs, especially those related to memory system resources.

These works rely on real system characterizations and simulation to develop a rigorous understanding of the interference and bottlenecks, and to provide solutions. Our analyses have shown key scaling and performance bottlenecks, proposed new solutions, and have inspired the research community to develop further investigations (e.g., on interference and fairness in main memory [41, 42, 43, 45], subarray-level parallelism [8, 13], low-cost memory reliability [21], hybrid memory management [20, 22, 23, 32, 50]). In order to aid future research, we have released our flexible and extensible memory system simulator, Ramulator, as open-source software [15, 38], and have released open-source simulators that accurately model memory interference in multicore systems [34, 36] and memory resource bottlenecks in GPU systems [35, 37].

In each work that is featured in this issue, based on our rigorous studies and analyses, we propose novel solutions that mitigate many of these problems. We examine GPUs as a special example because they enable massively parallel processing on a single chip and, as a result, are limited greatly by the bottlenecks in the memory system. For each of the works presented in this special issue, its corresponding article examines the work's significance in the context of modern computer systems, and discusses several new research questions and directions that each work motivates.

We start with three of our works that manage interference and contention in main memory. When multiple applications (or multiple threads of one or more applications) concurrently issue memory requests, these requests often contend with each other in the main memory system, increasing the average memory access latency and reducing per-application or per-thread parallelism. This contention becomes especially problematic when a highly-memory-intensive application issues many more requests than other applications, causing requests from the other applications to unfairly wait for very long times as the memory system takes time to service all of the requests from the highly-memory-intensive application. To mitigate the interference that each application induces on the other applications, memory systems must adopt new mechanisms to regulate the available memory bandwidth among all applications and/or reduce the amount of memory-level contention. Doing so can enable systems that are higher performance, more predictable, and more energy efficient at the same time. The first three works featured in this issue enable new mechanisms to more efficiently manage interference and contention in main memory.

The first paper in the issue describes Memory Interference-induced Slowdown Estimation (MISE), which originally appeared in HPCA 2013 [43]. This work (1) develops a model called MISE, which predicts the impact of interference in DRAM on the overall system performance; and (2) uses this model to design new memory schedulers that improve fairness and QoS among concurrently-executing applications. The work finds that various MISE-based memory schedulers can (1) provide predictable performance to designated applications and (2) significantly improve the overall system throughput.

The second paper in the issue describes Staged Memory Scheduling, which originally appeared in ISCA 2012 [3]. This work analyzes the high impact of interference between the CPU and GPU in a heterogeneous system (e.g., a system-on-chip), showing that the GPU can overwhelm CPU performance and sometimes vice versa. Based on this finding, the work develops a new memory controller that provides fair memory access for both CPU and GPU applications, improving the performance of CPU applications without affecting the throughput of GPU applications.

The third paper in the issue describes Subarray-Level Parallelism (SALP), which originally appeared in ISCA 2012 [13]. This work exploits the subarrays (i.e., sub-banking) in DRAM architectures to greatly increase the amount of memory parallelism available to applications. SALP proposes three new mechanisms to expose the subarrays to the memory controller at low cost, improving row locality and reducing the number of high-latency bank conflicts that occur when multiple requests access the same memory bank. The reduced bank conflicts and the improved row locality significantly improve overall system performance and reduce energy consumption.

Next, we look at several of our works that address the growing overheads and expenses associated with growing main memory densities and latencies. As systems execute more applications in parallel, and as applications process larger amounts of data, DRAM manufacturers have relied on aggressive technology scaling to increase the density of each DRAM device. Unfortunately, such scaling has introduced a number of key challenges [25, 28, 29], which we methodically address in the next four works.

Our fourth paper in the issue describes DSARP, which originally appeared in HPCA 2014 [8]. This work explores how increasing memory density will cause DRAM refresh operations to become a bigger performance bottleneck, preventing the DRAM from effectively servicing outstanding memory requests with low latency. The work proposes new memory controller policies that almost completely eliminate the performance overhead of DRAM refresh by performing refresh operations in the background via low-cost changes to the DRAM architecture and the memory controller.

Our fifth paper in the issue describes ChargeCache, which originally appeared in HPCA 2016 [12]. This work finds that many applications must reopen memory rows soon after

they are closed because of interference (i.e., bank conflicts), incurring a high access latency. ChargeCache is a new mechanism that takes advantage of the high charge held within a recently-closed row to reduce the access latency to such a row when it is accessed again soon in the future. The work shows that ChargeCache significantly improves the overall system performance and energy consumption.

Our sixth paper in the issue describes heterogeneous-reliability memory (HRM), which originally appeared in DSN 2014 [21]. This work demonstrates on real machines that many data center applications can tolerate errors in large regions of their memory address spaces without affecting correctness. The work uses this observation to lower the cost of memory subsystems for data centers, by introducing a new memory system framework, HRM, where the memory system consists of different modules with different types and amounts of error correction/detection capabilities. By employing many DRAM modules without error correction and intelligently mapping error-tolerant memory regions to these modules and error-vulnerable memory regions to DRAM modules with error correction, HRM significantly reduces the cost of a data center system, while still providing high overall reliability and availability.

Our seventh paper in the issue describes row buffer locality aware (RBLA) caching, which originally appeared in ICCD 2012 [50]. This work proposes a new technique to manage data placement in hybrid memory systems, which combine conventional DRAM with emerging memory technologies to provide the benefits of both in a scalable yet cost-effective manner. Exploiting the key observation that row buffer hits are of the same cost in both DRAM and emerging memory technologies, RBLA avoids migrating data from the emerging memory to conventional DRAM (and vice versa) when the migration would not yield a significant benefit, thereby preserving the precious DRAM space for data that really benefits from the low access latency of DRAM arrays. The work shows that RBLA improves both system performance and energy consumption as a result.

Finally, we examine how to manage memory resources within GPUs. For many general-purpose GPU (GPGPU) applications, programmers are responsible for explicitly managing all memory resources, including registers, by specifying in programs how much each application should get of each resource. Our solutions automatically manage these resources in both hardware and software, and sometimes cooperatively between the hardware and software, transparently to the programmer. The solutions lift the burden of resource management from the programmer, and improve the performance and efficiency of GPGPU applications.

Our eighth paper in the issue describes Zorua, which originally appeared in MICRO 2016 [46]. Current GPU systems require programmers to discover and explicitly specify the quantities of each resource that are assigned to a thread, in order to avoid significant performance penalties. This work pro-

poses a new resource virtualization mechanism for GPGPU applications, called Zorua, which can assign resources to each thread dynamically at runtime based on the thread’s needs and the available resources in the GPU, with only annotations provided by the compiler. With its effective resource virtualization, Zorua improves (1) programmability, by removing the existing burden on programmers to tune the thread resource allocation; (2) portability, by removing the need to retune the resource allocation when an application tuned for one GPU architecture is executed on a different GPU architecture; and (3) performance, by ensuring the careful allocation and oversubscription of resources to best utilize the hardware.

Our ninth paper in the issue describes Memory Divergence Correction (MeDiC), which originally appeared in PACT 2015 [4]. This work finds that different warps (i.e., groups of threads that execute in lockstep) exhibit different levels of memory divergence, where some, but not all, threads stall on long-latency memory accesses, which prevents forward progress for all threads in the warp. MeDiC consists of three new mechanisms that work together to optimize cache and memory resource management in a GPU, based on the divergence behavior of the warps belonging to an application. These three mechanisms provide significant performance improvements for GPGPU applications.

Our tenth paper in the issue describes Mosaic, which originally appeared in MICRO 2017 [1]. In contemporary GPUs, limited resources for memory virtualization can cause a single operation (e.g., an address translation that misses in the GPU’s translation lookaside buffer) to often stall hundreds of threads for long latencies, leading to significant underutilization of the GPU. The memory virtualization bottleneck can be alleviated by changing the page size, but a major hurdle to this is the key trade-off between two costly operations: demand paging (which benefits from small page sizes) and address translation (which benefits from large page sizes). This work proposes a new hardware mechanism that takes advantage of GPGPU memory access patterns to enable the efficient support of multiple page sizes transparently to the programmer. By efficiently supporting multiple page sizes, Mosaic alleviates the high contention for memory virtualization resources, which in turn significantly improves the performance of GPGPU applications.

Throughout all of these works, we (1) identify various points of interference, contention, and resource bottlenecks in memory systems and GPUs; and (2) appropriately modify the systems to mitigate these issues at low cost and low overhead. These works improve the performance, fairness, energy consumption, and/or programmability of a system, and often improve scalability as more applications execute concurrently on the system. Even though the works presented are described in the context of DRAM, the dominant memory technology of today, we believe many of the basic ideas and concepts can be applied or adapted to emerging memory technologies [22], e.g., phase-change memory [17, 18, 19, 31, 48, 49, 51],

STT-MRAM [11, 16, 30], and memristors/RRAM [10, 40, 47]. We hope that the works featured in this special issue inspire readers to explore other sources of interference, contention, performance, and programmability issues in modern systems, and to develop new solutions that can enable fair, high-performance, energy-efficient systems for the future.

Acknowledgments

The works featured in this issue, along with our related works that we reference in each featured work, are a result of the research done together with many students and collaborators over the course of the past 10+ years, whose contributions we acknowledge. In particular, we acknowledge and appreciate the dedicated effort of current and former students and postdocs in our research group, SAFARI [33, 39], who contributed to the ten featured works, including Kevin Chang, Rachael Harding, Hasan Hassan, Kevin Hsieh, Ben Jaiyen, Samira Khan, Yoongu Kim, Donghyuk Lee, Yixin Luo, Justin Meza, Gennady Pekhimenko, Vivek Seshadri, Ashish Shrestha, Lavanya Subramanian, Nandita Vijaykumar, and HanBin Yoon.

Aside from our featured works and other referenced papers from our group, where a wealth of information on modern memory and storage systems can be found, at least four Ph.D. dissertations have shaped the works that we feature in this special issue:

- Lavanya Subramanian’s thesis entitled “Providing High and Controllable Performance in Multicore Systems Through Shared Resource Management” [44].
- Yoongu Kim’s thesis entitled “Architectural Techniques to Enhance DRAM Scaling” [14].
- Kevin Chang’s thesis entitled “Understanding and Improving the Latency of DRAM-Based Memory Systems” [9], and
- Rachata Ausavarungnirun’s thesis entitled “Techniques for Shared Resource Management in Systems with Throughput Processors” [2].

We also acknowledge various funding agencies (the National Science Foundation, the Semiconductor Research Corporation, the Intel Science and Technology Center on Cloud Computing, CyLab, the CMU Data Storage Systems Center, and the NIH) and industrial partners (AMD, Facebook, Google, HP Labs, Huawei, IBM, Intel, Microsoft, NVIDIA, Oracle, Qualcomm, Rambus, Samsung, Seagate, VMware), and ETH Zürich, who have supported the featured works in this issue and other related work in our research group generously over the years.

References

- [1] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu, “Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes,” in *MICRO*, 2017.
- [2] R. Ausavarungnirun, “Techniques for Shared Resource Management in Systems with Throughput Processors,” Ph.D. dissertation, Carnegie Mellon Univ., 2017.
- [3] R. Ausavarungnirun, K. K. Chang, L. Subramanian, G. H. Loh, and O. Mutlu, “Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems,” in *ISCA*, 2012.

- [4] R. Ausavarungnirun, S. Ghose, O. Kayiran, G. H. Loh, C. R. Das, M. T. Kandemir, and O. Mutlu, "Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance," in *FACT*, 2015.
- [5] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, and O. Mutlu, "Error Characterization, Mitigation, and Recovery in Flash-Memory-Based Solid-State Drives," *Proc. IEEE*, 2017.
- [6] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, and O. Mutlu, "Error Characterization, Mitigation, and Recovery in Flash Memory Based Solid-State Drives," arXiv:1706.08642 [cs.AR], 2017.
- [7] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, and O. Mutlu, "Errors in Flash-Memory-Based Solid-State Drives: Analysis, Mitigation, and Recovery," arXiv:1711.11427 [cs.AR], 2017.
- [8] K. K. Chang, D. Lee, Z. Chishti, A. R. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu, "Improving DRAM Performance by Parallelizing Refreshes with Accesses," in *HPCA*, 2014.
- [9] K. K. Chang, "Understanding and Improving the Latency of DRAM-Based Memory Systems," Ph.D. dissertation, Carnegie Mellon Univ., 2017.
- [10] L. Chua, "Memristor—The Missing Circuit Element," *TCT*, 1971.
- [11] X. Guo, E. Ipek, and T. Soyata, "Resistive Computation: Avoiding the Power Wall with Low-Leakage, STT-MRAM Based Computing," in *ISCA*, 2010.
- [12] H. Hassan, G. Pekhimenko, N. Vijaykumar, V. Seshadri, D. Lee, O. Ergin, and O. Mutlu, "ChargeCache: Reducing DRAM Latency by Exploiting Row Access Locality," in *HPCA*, 2016.
- [13] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM," in *ISCA*, 2012.
- [14] Y. Kim, "Architectural Techniques to Enhance DRAM Scaling," Ph.D. dissertation, Carnegie Mellon Univ., 2015.
- [15] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," *CAL*, 2015.
- [16] E. Kültürsay, M. Kandemir, A. Sivasubramanian, and O. Mutlu, "Evaluating STT-RAM as an Energy-Efficient Main Memory Alternative," in *ISPASS*, 2013.
- [17] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting Phase Change Memory as a Scalable DRAM Alternative," in *ISCA*, 2009.
- [18] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Phase Change Memory Architecture and the Quest for Scalability," *CACM*, 2010.
- [19] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-Change Technology and the Future of Main Memory," *IEEE Micro*, 2010.
- [20] Y. Li, S. Ghose, J. Choi, J. Sun, H. Wang, and O. Mutlu, "Utility-Based Hybrid Memory Management," in *CLUSTER*, 2017.
- [21] Y. Luo, S. Govindan, B. Sharma, M. Santaniello, J. Meza, A. Kansal, J. Liu, B. Khes-sib, K. Vaid, and O. Mutlu, "Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-Reliability Memory," in *DSN*, 2014.
- [22] J. Meza, Y. Luo, S. Khan, J. Zhao, Y. Xie, and O. Mutlu, "A Case for Efficient Hardware-Software Cooperative Management of Storage and Memory," in *WEED*, 2013.
- [23] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan, "Enabling Efficient and Scalable Hybrid Memories Using Fine-Granularity DRAM Cache Management," *CAL*, 2012.
- [24] T. Moscibroda and O. Mutlu, "Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems," in *USENIX Security*, 2007.
- [25] O. Mutlu, "The RowHammer Problem and Other Issues We May Face as Memory Becomes Denser," in *DATE*, 2017.
- [26] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *MICRO*, 2007.
- [27] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems," in *ISCA*, 2008.
- [28] O. Mutlu, "Memory Scaling: A Systems Architecture Perspective," in *IMW*, 2013.
- [29] O. Mutlu and L. Subramanian, "Research Problems and Opportunities in Memory Systems," *SUPERFRI*, 2014.
- [30] H. Naeimi, C. Augustine, A. Raychowdhury, S.-L. Lu, and J. Tschanz, "STT-RAM Scaling and Retention Failure," *Intel Technology Journal*, 2013.
- [31] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable High Performance Main Memory System Using Phase-Change Memory Technology," in *ISCA*, 2009.
- [32] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu, "ThyNVM: Enabling Software-Transparent Crash Consistency in Persistent Memory Systems," in *MICRO*, 2015.
- [33] SAFARI Research Group, <http://www.ece.cmu.edu/~safari/>.
- [34] SAFARI Research Group, "ASMSim – GitHub Repository," <https://github.com/CMU-SAFARI/ASMSim>.
- [35] SAFARI Research Group, "MeDiC GPGPU-Sim Patch – GitHub Repository," <https://github.com/CMU-SAFARI/MeDiC>.
- [36] SAFARI Research Group, "MemSchedSim – GitHub Repository," <https://github.com/CMU-SAFARI/MemSchedSim>.
- [37] SAFARI Research Group, "Mosaic Simulator – GitHub Repository," <https://github.com/CMU-SAFARI/Mosaic>.
- [38] SAFARI Research Group, "Ramulator – GitHub Repository," <https://github.com/CMU-SAFARI/ramulator>.
- [39] SAFARI Research Group, "SAFARI Software Tools – GitHub Repository," <https://github.com/CMU-SAFARI/>.
- [40] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The Missing Memristor Found," *Nature*, 2008.
- [41] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, "The Blacklisting Memory Scheduler: Achieving High Performance and Fairness at Low Cost," in *ICCD*, 2014.
- [42] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, "BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling," *TPDS*, 2016.
- [43] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu, "MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems," in *HPCA*, 2013.
- [44] L. Subramanian, "Providing High and Controllable Performance in Multicore Systems Through Shared Resource Management," Ph.D. dissertation, Carnegie Mellon Univ., 2015.
- [45] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu, "The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory," in *MICRO*, 2015.
- [46] N. Vijaykumar, K. Hsieh, G. Pekhimenko, S. Khan, A. Shrestha, S. Ghose, A. Jog, P. B. Gibbons, and O. Mutlu, "Zorua: A Holistic Approach to Resource Virtualization in GPUs," in *MICRO*, 2016.
- [47] H.-S. P. Wong, H.-Y. Lee, S. Yu, Y.-S. Chen, Y. Wu, P.-S. Chen, B. Lee, F. T. Chen, and M.-J. Tsai, "Metal-Oxide RRAM," *Proc. IEEE*, 2012.
- [48] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase Change Memory," *Proc. IEEE*, 2010.
- [49] H. Yoon, J. Meza, N. Muralimanohar, N. P. Jouppi, and O. Mutlu, "Efficient Data Mapping and Buffering Techniques for Multi-Level Cell Phase-Change Memories," *TACO*, 2014.
- [50] H. Yoon, J. Meza, R. Ausavarungnirun, R. A. Harding, and O. Mutlu, "Row Buffer Locality Aware Caching Policies for Hybrid Memories," in *ICCD*, 2012.
- [51] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology," in *ISCA*, 2009.

Predictable Performance and Fairness Through Accurate Slowdown Estimation in Shared Main Memory Systems

Lavanya Subramanian^{1,2} Vivek Seshadri^{3,2}
Yoongu Kim² Ben Jaiyen^{4,2} Onur Mutlu^{5,2}

¹Intel Labs ²Carnegie Mellon University ³Microsoft Research India ⁴Google ⁵ETH Zürich

This paper summarizes the ideas and key concepts of MISE (Memory Interference-induced Slowdown Estimation), which was published in HPCA 2013 [97], and examines the work’s significance and future potential. Applications running concurrently on a multicore system interfere with each other at the main memory. This interference can slow down different applications differently. Accurately estimating the slowdown of each application in such a system can enable mechanisms that can enforce quality-of-service. While much prior work has focused on mitigating the performance degradation due to inter-application interference, there is little work on accurately estimating slowdown of individual applications in a multi-programmed environment. Our goal is to accurately estimate application slowdowns, towards providing predictable performance.

To this end, we first build a simple Memory Interference-induced Slowdown Estimation (MISE) model, which accurately estimates slowdowns caused by memory interference. We then leverage our MISE model to develop two new memory scheduling schemes: 1) one that provides soft quality-of-service guarantees, and 2) another that explicitly attempts to minimize maximum slowdown (i.e., unfairness) in the system. Evaluations show that our techniques perform significantly better than state-of-the-art memory scheduling approaches to address the above problems.

Our proposed model and techniques have enabled significant research in the development of accurate performance models [35, 59, 98, 110] and interference management mechanisms [66, 66, 99, 100, 108, 119, 120].

1. Problem: Unpredictable Slowdowns

In a multicore system, multiple applications are consolidated on the same machine. While consolidation may enable better resource utilization, it results in interference between applications at the shared resources, slowing down each application to a different degree. Specifically, main memory is a heavily contended shared resource between applications in a multicore system. Each application accessing the memory experiences different and unpredictable slowdowns depending on the available memory bandwidth and the other concurrently running applications.

A large body of work proposed several different approaches to mitigate memory interference between applications with the goal of improving overall system performance. This includes memory scheduling [2, 18, 27, 32, 42, 43, 50, 72, 76, 77, 80, 99, 100, 103, 117], memory channel/bank partitioning [36, 64, 74],

memory interleaving [38], source throttling [3, 7, 17, 19, 102], and thread scheduling [14, 101, 106, 121] techniques. However, few previous works (notably [15, 17, 19, 76]) have attempted to estimate individual application slowdowns online with the goal of providing predictable performance.

Our goal in our HPCA 2013 paper [97] is to provide predictable performance for individual applications. To this end, we first design a model to accurately estimate memory-interference-induced slowdowns of applications running concurrently on a multicore system. We then leverage this model to design effective mechanisms to enforce quality-of-service (QoS) and achieve fairness.

2. The Memory Interference-Induced Slowdown Estimation (MISE) Model

The slowdown of an application indicates the performance of the application, when it is sharing resources with other applications, relative to when the application is run alone. Slowdown can be expressed as

$$\text{Slowdown of an App.} = \frac{\text{alone-performance}}{\text{shared-performance}} \quad (1)$$

Hence, estimating the slowdown of an individual application requires two pieces of information: 1) the performance of the application when it is run concurrently with other applications (i.e., *shared-performance*), and 2) the performance of the application when it is run alone on the same system (i.e., *alone-performance*). While the former can be directly measured, the key challenge is to estimate the performance the application would have if it were running alone *while* it is actually running alongside other applications. This requires quantifying the effect of interference on application performance.

2.1. Key Observations

In this work, we make two observations that lead to a simple and effective model to estimate the slowdown of individual applications.

Observation 1: *The performance of a memory-bound application is roughly proportional to the rate at which its memory requests are served.* This observation stems from a memory-bound application’s characteristic to spend an overwhelmingly large fraction of its execution time stalling on

memory accesses. Therefore, the rate at which such an application's requests are served has significant impact on its performance.

To validate this observation, we conducted a real-system experiment where we ran a memory-bound application from the SPEC CPU2006 benchmark suite [96] alongside three copies of a microbenchmark whose memory intensity can be varied, on a 4-core Intel Core i7 [31].¹ By varying the memory intensity, i.e., the last-level cache (LLC) miss rate, of the microbenchmark, we can change the rate at which the requests of the SPEC application are served. Figure 1 plots the results of this experiment for three memory-intensive benchmarks, *mcf*, *omnetpp*, and *astar*. The figure shows the performance of each application versus the rate at which its requests are served.

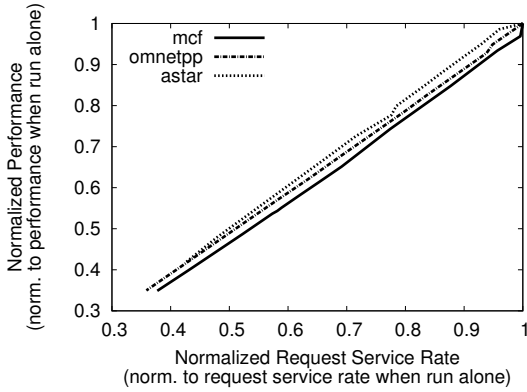


Figure 1: Request service rate vs. performance. Reproduced from [97].

The results of this experiment validate our observation. The performance of a memory-bound application is directly proportional to the rate at which its requests are served. This suggests that we can use the request-service-rate of an application as a proxy for its performance. More specifically, we can estimate the slowdown of an application, i.e., the ratio of its performance when it is run alone on a system vs. its performance when it is run alongside other applications on the same system, as follows:

$$\text{Slowdown of an App.} = \frac{\text{alone-request-service-rate}}{\text{shared-request-service-rate}} \quad (2)$$

Estimating the *shared-request-service-rate* (SRSR) of an application is straightforward. It only requires the memory controller to keep track of how many requests of the application are served in a given number of cycles. However, the challenge is to estimate the *alone-request-service-rate* (ARSR) of an application *while* it is run alongside other applications. A naive way of estimating ARSR of an application would be to prevent all other applications from accessing memory for

¹The microbenchmark streams through a large region of memory (one block at a time). The memory intensity of the microbenchmark (last-level cache misses per kilo-instruction, i.e., LLC MPKI) is varied by changing the amount of computation performed between memory operations.

a length of time and measure the application's ARSR. While this approach might provide an estimate of the application's ARSR, it would significantly slow down other applications in the system and is prone to incorrect estimations due to phase fluctuations in the application. Our second observation helps us to address this problem.

Observation 2: *The ARSR of an application can be estimated by giving the requests of the application the highest priority in accessing memory.*

Giving an application's requests the *highest priority* in accessing memory results in very little interference from the requests of other applications. Therefore, requests of the application are served almost as if the application were the only one running on the system. Based on the above observation, the ARSR of an application can be estimated as:

$$\text{ARSR of an App.} = \frac{\# \text{ Requests with Highest Priority}}{\# \text{ Cycles with Highest Priority}} \quad (3)$$

where *# Requests with Highest Priority* is the number of requests served when the application is given highest priority, and *# Cycles with Highest Priority* is the number of cycles an application is given highest priority by the memory controller.

The memory controller can use Equation 3 to periodically estimate the ARSR of an application. We add an interference counter to capture the remaining interference cycles. The details of the mechanisms we add to increase the accuracy of the model are described in Section 4 of our HPCA 2013 paper [97]. Once we estimate ARSR, Equation 2 can be used to estimate the slowdown of the application.

2.2. MISE Model for Non-Memory-Bound Applications

So far, we have described the key observations of the MISE model for a memory-bound application. We find that the model presented above has low accuracy for non-memory-bound applications. This is because a non-memory-bound application spends a significant fraction of its execution time in the *compute phase* (when the core is *not* stalled waiting for memory). Hence, varying the request service rate for such an application will not affect the length of the large compute phase. Therefore, we take into account the duration of the compute phase to make the model accurate for non-memory-bound applications.

Let α be the fraction of time spent by an application stalling at memory. Therefore, the fraction of time spent by the application in the compute phase is $1 - \alpha$. Since changing the request service rate affects only the memory phase, we augment Equation 2 to take into account α as follows:

$$\text{Slowdown of an App.} = (1 - \alpha) + \alpha \frac{\text{ARSR}}{\text{SRSR}} \quad (4)$$

In addition to estimating ARSR and SRSR required by Equation 2, the above equation requires estimating the parameter α , the fraction of time spent in the memory phase. However,

precisely computing α for a modern out-of-order processor is a challenge since such a processor overlaps computation with memory accesses. The processor stalls waiting for memory only when the oldest instruction in the reorder buffer is waiting on a memory request. For this reason, we estimate α as the fraction of time the processor spends stalling for memory:

$$\alpha = \frac{\text{\# Cycles spent stalling on memory requests}}{\text{Total number of cycles}} \quad (5)$$

More details of our MISE slowdown estimation model are described in Sections 3 and 4 of our HPCA 2013 paper [97]. More recently, we used this model to expand slowdown estimation to a memory hierarchy that also includes shared caches, as part of the Application Slowdown Model [98].

3. Evaluation of the MISE Model

We compare the MISE model against the slowdown estimation model employed by the Stall Time Fair Memory Scheduler (STFM) [76], which is the closest previous work on estimating memory interference-induced slowdown.² STFM estimates the slowdown of an application by estimating the number of cycles it stalls due to interference from other applications' requests. In this section, we qualitatively and quantitatively compare MISE with STFM.

There are two key differences between MISE and STFM in estimating slowdown. First, MISE uses request service rates rather than stall times to estimate slowdown. In MISE, the *alone-request-service-rate* of an application can be fairly accurately estimated by giving the application highest priority in accessing memory. Giving the application highest priority in accessing memory results in very little interference from other applications. In contrast, STFM attempts to estimate the alone-stall-time of an application while it is receiving significant interference from other applications, which turns out to be difficult to do accurately. Second, MISE takes into account the effect of the compute phase for non-memory-bound applications. STFM, on the other hand, has no such provision to account for the compute phase. As a result, MISE's slowdown estimates for non-memory-bound applications are significantly more accurate than STFM's estimates.

Figure 2 compares the accuracy of MISE with STFM for two representative memory-bound applications, lbm and les3d. Figure 3 compares the accuracy of MISE with STFM for two representative non-memory-bound applications, wrf and povray. Each of these applications is run on a 4-core system with three other applications. Our detailed experimental methodology is provided in Section 5 of our HPCA 2013 paper [97]. This includes detailed descriptions of our experimental setup, workloads and metrics. Furthermore,

²FST [17] and Du Bois et al.'s per-thread cycle accounting mechanism [15] are the other two previous works that estimate application slowdown. The mechanism to estimate main memory interference induced slowdown in both of these previous works is similar to STFM.

our simulator implementing the MISE model is available online [90]. As can be observed, MISE's slowdown estimates are much closer to the actual slowdown than STFM's estimates. This is because the MISE model eliminates a significant portion of the interference received by an application while estimating slowdown, by prioritizing it in the memory controller. On the other hand, STFM estimates slowdown *while* an application is experiencing interference.

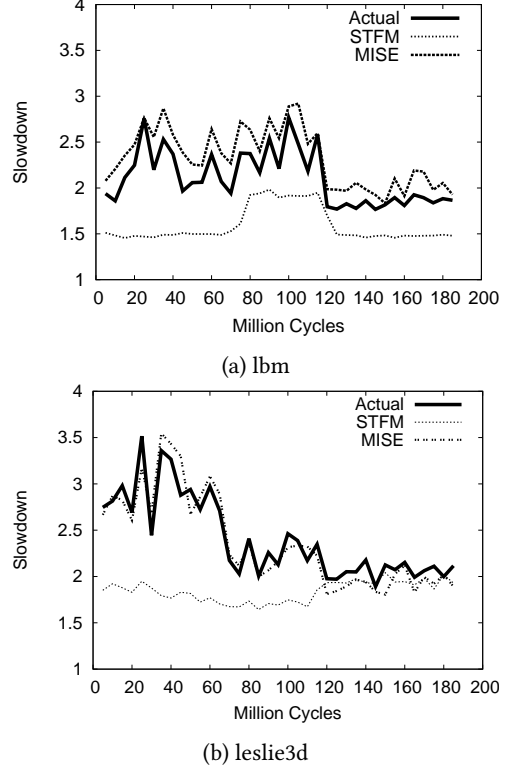


Figure 2: Comparison of MISE with STFM for representative memory-bound applications. Adapted from [97].

Table 1 shows the average slowdown estimation error for each benchmark, with STFM and MISE, across 300 4-core workloads of different memory intensities. As can be observed, MISE's slowdown estimates have significantly lower error than STFM's slowdown estimates across most benchmarks. Across 300 workloads, STFM's estimates deviate from the actual slowdown by 29.8%, whereas, our proposed MISE model's estimates deviate from the actual slowdown by only 8.1%. Therefore, we conclude that our slowdown estimation model provides better accuracy than STFM.

For a more detailed analysis of the MISE model's accuracy and characteristics, we refer the reader to our HPCA 2013 paper [97].

4. Leveraging the MISE Model

Accurate slowdown estimates are a key enabler towards designing mechanisms to better enforce quality-of-service (QoS) and fairness. Slowdown estimates from the MISE model could be leveraged in hardware to design memory scheduling

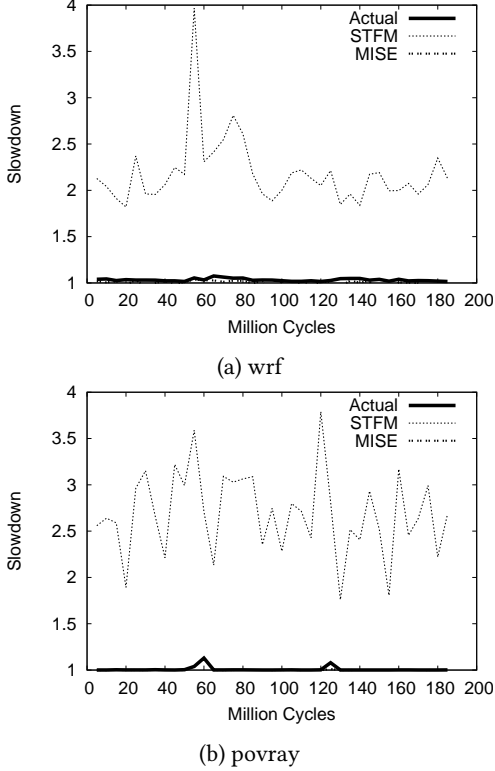


Figure 3: Comparison of MISE with STFM for representative non-memory-bound applications. Adapted from [97].

Table 1: Average slowdown estimation error for each benchmark (in %). Adapted from [97].

Benchmark	STFM	MISE	Benchmark	STFM	MISE
453.povray	56.3	0.1	473.astar	12.3	8.1
454.calculix	43.5	1.3	456.hmmmer	17.9	8.1
400.perlbench	26.8	1.6	464.h264ref	13.7	8.3
447.dealII	37.5	2.4	401.bzip2	28.3	8.5
436.cactusADM	18.4	2.6	458.sjeng	21.3	8.8
450.soplex	29.8	3.5	433.milc	26.4	9.5
444.namd	43.6	3.7	481.wrf	33.6	11.1
437.leslie3d	26.4	4.3	429.mcf	83.74	11.5
403.gcc	25.4	4.5	445.gobmk	23.1	12.5
462.libquantum	48.9	5.3	483.xalancbmk	18.0	13.6
459.GemsFDTD	21.6	5.5	435.gromacs	31.4	15.6
470.lbm	6.9	6.3	482.sphinx3	21	16.8
473.astar	12.3	8.1	471.omnetpp	26.2	17.5
456.hmmmer	17.9	8.1	465.tonto	32.7	19.5

policies to provide QoS guarantees and fairness. Alternatively, the slowdown estimates could be communicated to the system software, which could leverage them to perform application scheduling, admission control and migration. We will describe two such mechanisms that leverage the MISE model: 1) MISE-QoS, a mechanism to provide soft QoS guarantees in the memory controller; and 2) MISE-Fair, a mechanism to minimize maximum slowdown [13, 14, 42, 43, 92, 99, 100, 103] to improve overall system fairness.

4.1. MISE-QoS: Providing Soft QoS Guarantees

MISE-QoS aims to provide soft slowdown guarantees to an application of interest (AoI) in a workload with many applications, while trying to maximize overall performance for the remaining applications. There are two aspects of providing a soft slowdown guarantee. One is to ensure that the application of interest is not slowed down by more than a system-software-specified bound. The other aspect is to detect if the bound is *not met* for some reason.

MISE-QoS addresses both of these aspects by using slowdown estimates from the MISE model. It periodically obtains slowdown estimates from the MISE model and increases/decreases the memory bandwidth allocated to the AoI such that the AoI receives just enough bandwidth to meet its slowdown bound. This enables the other applications to use the remaining bandwidth, improving their performance. MISE-QoS addresses the second aspect by comparing slowdown estimates from the MISE model with the prescribed bound periodically. When the prescribed bound cannot be met despite always prioritizing the AoI, MISE-QoS detects that the bound cannot be met just by prioritizing the application at the memory controller.

Previous work [34] attempts to address the first aspect by *always* prioritizing the AoI. This may unnecessarily slowdown other applications in the system by excessively prioritizing the AoI, especially when the AoI is meeting its performance bound. Furthermore, such a mechanism, in the absence of accurate slowdown estimates, does not have the provision to detect whether or not the bound is met.

Slowdown Evaluation. We evaluate the MISE-QoS mechanism across 300 workloads with 10 different slowdown bounds for each workload. Our results show that the MISE-QoS mechanism meets the prescribed slowdown bound for 97.5% of the workloads for which the naive mechanism that always prioritizes the AoI meets the bound, while improving overall system performance by 12%. Furthermore, MISE-QoS also predicts whether or not the bound is met with an accuracy of 95.7%, while previous work [34] has no such provision.

To show the effectiveness of MISE-QoS, we compare the AoI's slowdown due to MISE-QoS and the mechanism that always prioritizes the AoI (*Always Prioritize*) [34]. Figure 4 presents representative results for 8 different AoIs when they are run alongside three other applications. The label MISE-QoS-n corresponds to a slowdown bound of $\frac{10}{n}$. (Note that *Always Prioritize* does not take into account the slowdown bound.) Note that the slowdown bound decreases (i.e., becomes tighter) from left to right for each benchmark in Figure 4 (as well as in other figures).

We draw three conclusions from the results. First, for most applications, the slowdown of *Always Prioritize* is considerably more than one. This indicates that always prioritizing the AoI does not completely prevent other applications from interfering with the AoI. Second, as the slowdown bound for the AoI is decreased (left to right), MISE-QoS gradually increases

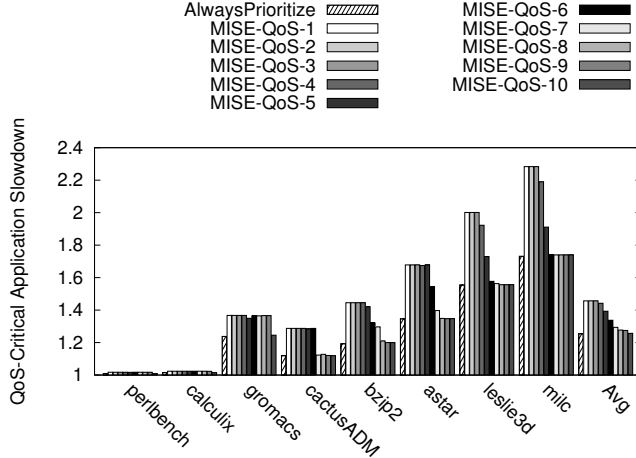


Figure 4: AoI performance: MISE-QoS vs. *AlwaysPrioritize*. Reproduced from [97].

the bandwidth allocation for the AoI, eventually allocating all the available bandwidth to the AoI. At this point, MISE-QoS performs very similarly to the *Always Prioritize* mechanism. Third, in almost all cases (in this figure and across all our 3000 data points), MISE-QoS meets the specified slowdown bound *if* *Always Prioritize* is able to meet the bound (see Section 8.1 of our HPCA 2013 paper [97] for details).

System Performance and Fairness. Figure 5 compares the system performance (harmonic speedup) and fairness (maximum slowdown) of MISE-QoS and *Always Prioritize* for different values of the bound. We omit the AoI from the performance and fairness calculations. The results are categorized into four workload categories (0, 1, 2, 3) indicating the number of memory-intensive benchmarks in the workload. For clarity, the figure shows results only for a few slowdown bounds. Three conclusions are in order.

First, MISE-QoS significantly improves performance compared to *Always Prioritize*, especially when the slowdown bound for the AoI is large. On average, when the bound is $\frac{10}{3}$, MISE-QoS improves harmonic speedup [67] by 12% and weighted speedup [22, 95] by 10% (not shown due to lack of space) over *Always Prioritize*, while reducing maximum slowdown [13, 14, 42, 43, 92, 99, 100, 103] by 13%. Second, as expected, the performance and fairness of MISE-QoS approach that of *Always Prioritize* as the slowdown bound is decreased (going from left to right for a set of bars). Finally, the benefits of MISE-QoS increase with increasing memory intensity because always prioritizing a memory intensive application will cause significant interference to other applications.

Based on our results, we conclude that MISE-QoS can effectively ensure that the AoI meets the specified slowdown bound while achieving high system performance and fairness across the other applications.

4.2. MISE-Fair: Minimizing Maximum Slowdown

The second mechanism we build on top of our MISE model is one that seeks to improve overall system fairness. Specifi-

cally, this mechanism attempts to minimize the maximum slowdown across all applications in the system. Ensuring that no application is unfairly slowed down while maintaining high system performance is an important goal in multi-core systems where co-executing applications are similarly important. Many prior works evaluate fairness in such scenarios in terms of the maximum slowdown of any application [13, 14, 42, 43, 92, 99, 100, 103].

At a high level, our mechanism works as follows. The memory controller maintains two pieces of information: 1) a target slowdown bound (B) for *all* applications, and 2) a bandwidth allocation policy that partitions the available memory bandwidth across all applications. The memory controller enforces the bandwidth allocation policy using a lottery-scheduling technique proposed in [105]. The controller attempts to ensure that the slowdown of all applications is within the bound B . To this end, it modifies the bandwidth allocation policy so that applications that are slowed down more get more memory bandwidth. Should the memory controller find that bound B is not possible to meet, it increases the bound. On the other hand, if the bound is easily met, it decreases the bound.

Interaction with the Operating System. As we will show in Section 4.2, our mechanism provides the best fairness compared to three state-of-the-art approaches for memory request scheduling [42, 43, 76]. In addition to this, there is another benefit to using our approach. Our mechanism, based on the MISE model, can accurately estimate the slowdown of each application. Therefore, the memory controller can potentially communicate the estimated slowdown information to the operating system (OS). The OS can use this information to make more informed scheduling and mapping decisions in order to further improve system performance or fairness. Since prior memory scheduling approaches do *not* explicitly attempt to minimize maximum slowdown by accurately estimating the slowdown of individual applications, such a mechanism to interact with the OS is *not* possible with them. Evaluating the benefits of the interaction between our mechanism and the OS is beyond the scope of this paper but is an important area of future work.

Evaluation. Figure 6 compares the system fairness (maximum slowdown) of different mechanisms with increasing number of cores. The figure shows results with four previously proposed memory scheduling policies (FRFCFS [89, 122], ATLAS [42], TCM [43], and STFM [76]), and our proposed mechanism using the MISE model (MISE-Fair). We draw three conclusions from our results.

First, MISE-Fair provides the best fairness compared to all other previous approaches. The reduction in the maximum slowdown due to MISE-Fair when compared to STFM (the best previous mechanism) increases with increasing number of cores. With 16 cores, MISE-Fair provides 7.2% better fairness compared to STFM.

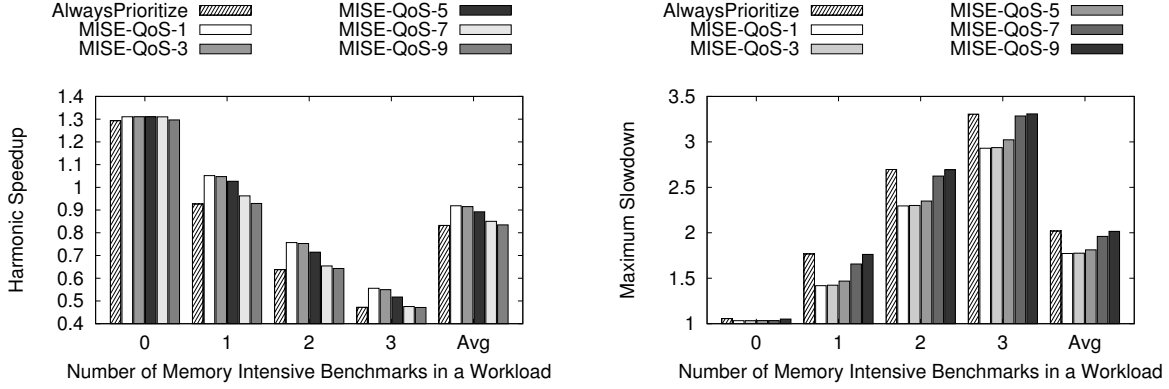


Figure 5: Average system performance and fairness across 300 workloads of different memory intensities. Reproduced from [97].

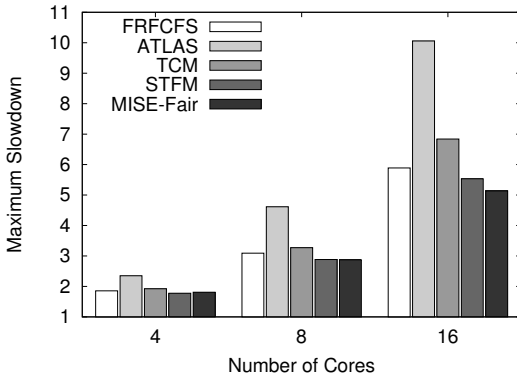


Figure 6: Fairness with different core counts. Reproduced from [97].

Second, STFM, as a result of prioritizing the most slowed down application, provides better fairness than all other previous approaches. While the slowdown estimates of STFM are not as accurate as those of our mechanism, they are good enough to identify the most slowed down application. However, as the number of concurrently-running applications increases, simply prioritizing the most slowed down application may not lead to better fairness. MISE-Fair, on the other hand, works towards reducing maximum slowdown by stealing bandwidth from those applications that are less slowed down compared to others. As a result, the fairness benefits of MISE-Fair compared to STFM increase with increasing number of cores.

Third, ATLAS and TCM are more unfair compared to FRFCFS. As shown in prior work [42, 43], ATLAS trades off fairness to obtain better performance. TCM, on the other hand, is designed to provide high system performance and fairness. Further analysis showed us that the cause of TCM’s unfairness is the strict ranking employed by TCM. TCM ranks all applications based on its clustering and shuffling techniques [43] and strictly enforces these rankings. We found that such strict ranking destroys the row-buffer locality of low-

ranked applications. This increases the slowdown of such applications, leading to high maximum slowdown.³

We conclude that the MISE model’s slowdown estimates can be used to design a better and more fair memory scheduler. We expect future works can take advantage of the MISE model to design even better memory scheduling and other resource management mechanisms.

5. Related Work

To our knowledge, this is the first paper to 1) provide a *simple and accurate* model to estimate application slowdowns in the presence of main memory interference, and 2) use this model to devise two new memory scheduling techniques that either aim to satisfy slowdown bounds of applications or improve system fairness and performance. In this section, we discuss several related works. We discuss works that build upon MISE in Section 6.1.

Slowdown Estimation. Stall Time Fair Memory Scheduling (STFM) [76] attempts to estimate each application’s slowdown, with the goal of improving fairness by prioritizing the most slowed down application. STFM estimates an application’s slowdown as the ratio of its memory stall time when it is run alone versus when it is concurrently run alongside other applications. The challenge is in determining the alone stall time of an application *while* the application is actually running alongside other applications. STFM proposes to address this challenge by counting the number of cycles an application is stalled due to interference from other applications at the DRAM channels, banks and row-buffers. STFM uses this interference cycle count to estimate the alone-stall-time of the application, and hence the application’s slowdown.

Fairness via Source Throttling (FST) [17] estimates application slowdowns due to inter-application interference at the shared caches and memory, as the ratio of uninterfered to interfered execution times. FST uses the slowdown estimates to make informed source throttling decisions, to improve fairness. The mechanism to account for memory interfe-

³Note that this observation later led us to develop the Blacklisting Memory Scheduler (BLISS) [99, 100].

rence to estimate uninterfered execution time is similar to that employed in STFM. Prefetch-Aware Shared Resource Management [19] extends the FST model to take into account prefetch requests.

A concurrent work by Du Bois et al. [15] proposes per-thread cycle accounting (PTCA) for multicore processors, which determines an application’s standalone execution time when it shares cache and memory with other applications in a multicore system. In order to quantify memory interference, PTCA counts the number of waiting cycles due to inter-application interference and factors out these waiting cycles to estimate alone execution times, which is similar to STFM’s alone stall time estimation mechanism.

Eyerman and Eeckhout [23] and Cazorla et al. [5] propose mechanisms to determine an application’s slowdown while it is running alongside other applications on an SMT processor. Luque et al. [68] estimate application slowdowns in the presence of shared cache interference. Lin and Balasubramanian [60] propose a regression-based model to estimate performance for different cache allocations. None of these studies take into account inter-application interference at the main memory. Therefore, MISE, which estimates slowdown due to main memory interference, can be combined with the above approaches to quantify interference at the SMT processor and shared cache to build a comprehensive mechanism.

Quality-of-Service (QoS). Several prior works provide QoS guarantees in shared memory CMP systems. Mars et al. [69] propose a mechanism to estimate an application’s sensitivity towards interference and its propensity to cause interference. They utilize this knowledge to make informed mapping decisions between applications and cores. However, this mechanism 1) assumes *a priori* knowledge of applications, which may not always be possible to have, and 2) is designed for only 2 cores, and it is not clear how it can be extended to more than 2 cores. In contrast, MISE does not assume any *a priori* knowledge of applications and works well with large core counts, as we have shown in this paper. That said, MISE can possibly be used to provide feedback to the mapping mechanism proposed by [69] to overcome the shortcomings of their mechanism.

Iyer et al. [30, 33, 34] propose mechanisms to provide guarantees on shared cache space, memory bandwidth or IPC for different applications. The slowdown guarantee provided by MISE-QoS is stricter than these mechanisms as MISE-QoS takes into account the alone-performance of each application. Nesbit et al. [80] propose a mechanism to enforce a bandwidth allocation policy, by partitioning the available bandwidth across concurrently running applications based on some policy. While we use a scheduling technique similar to lottery-scheduling [85, 105] to enforce the bandwidth allocation policies of MISE-QoS and MISE-Fair, the mechanism proposed by Nesbit et al. can also be used in our proposal to

allocate bandwidth instead of our lottery-scheduling approach.

Memory Interference Mitigation. Many prior works focus on the problem of mitigating inter-application interference at the main memory to improve system performance and/or fairness. Most of these approaches address memory interference by modifying the memory request scheduling algorithm [2, 18, 27, 32, 34, 42, 43, 50, 51, 52, 53, 72, 73, 76, 77, 80, 99, 100, 115, 117]. We quantitatively compare MISE-Fair to STFM [76], ATLAS [42], and TCM [43] in Section 4.2, and show that MISE-Fair provides better fairness than these prior approaches.

Other works examine approaches such as sub-row interleaving [38], channel/bank partitioning [36, 64, 74, 109], bandwidth partitioning [61, 97], source throttling [3, 7, 17, 19, 39, 81, 82, 102], thread scheduling [14, 101, 106, 121], and changes to DRAM design [44, 58]. These approaches are complementary to MISE, and can be combined to achieve better fairness.

Prior Work on Analytical Performance Modeling. Prior works attempt to quantify the impact of cache/memory contention through offline profiling. Mars et al. [69] estimate an application’s sensitivity/propensity to receive/cause interference. Other previous works propose to estimate an application’s sensitivity to cache capacity [20, 91] and memory bandwidth [21] through profiling. Yang et al. [111] attempt to estimate applications’ sensitivity to interference online. However, this work assumes that latency-critical applications run alone at times, when they can be profiled (which could degrade system throughput). These works assume the ability to profile (1) entire applications offline; or (2) specific execution scenarios, such as an application executing alone. In contrast, MISE can estimate the slowdown of any application *online*, in the general scenario of multiple applications running together.

Several previous works [24, 25, 37, 104] propose analytical models to estimate processor performance, as an alternative to time consuming simulations. The goal of our MISE model, in contrast, is to estimate slowdowns at runtime, in order to enable mechanisms to provide QoS and high fairness. Its use in simulation is possible, but is left to future work.

6. Significance

To our knowledge, our HPCA 2013 paper [97] is the first to build a *simple yet accurate* hardware-based model to estimate application slowdowns due to main memory interference online *with the goal of providing predictable performance*. Previous works [15, 17, 19, 76] propose mechanisms to estimate application slowdowns. However, these mechanisms are not accurate enough (as we demonstrate in Section 3) since they were not designed with the goal of providing predictable performance. Rather, the slowdown estimates were used to make prioritization/throttling decisions to improve overall fairness.

This work is also the first to design a hardware-based mechanism to i) provide soft guarantees on slowdown for ap-

plications and ii) detect when a prescribed slowdown bound is not being met, by leveraging slowdown estimates from the MISE model, while also improving overall system performance. Previous work [34], in the absence of a model to accurately estimate application slowdowns, always prioritizes the application that needs guaranteed performance, degrading the performance of other co-running applications. Furthermore, previous work also does not have the provision to detect whether or not the prescribed slowdown bounds are being met (as we describe in Section 4).

6.1. Retrospective and Works Building on Our HPCA 2013 Paper

Adoption of the Principles of the MISE Model. The principles employed in the MISE model have been adopted towards slowdown estimation in several works that followed. The application slowdown model (ASM) [98], a follow-on work, builds on top of MISE’s memory slowdown estimation model and extended it to take into account shared cache interference. In doing so, ASM also addressed one of the major caveats of the MISE model, the estimation of slowdown for non-memory-intensive applications. While MISE has a mechanism to address the slowdown of non-memory-intensive applications, this mechanism relies on the estimation of the memory-bound fraction of an application. Estimating the fraction of an application’s execution that is memory bound, with high fidelity, is challenging. ASM addresses this challenge by applying the observation on request service rate as a proxy for performance at the input to the shared caches. This seamlessly enables slowdown estimation for applications with different memory and cache intensities/sensitivities. The ASM work shows that it can accurately estimate slowdowns with only 9.9% error across 100 workloads. We refer the reader to [98] for details.

A later work by Xiong et al. [110] proposes a slowdown estimation model that adopts the principle of giving an application highest priority in order to estimate its alone run behavior. This work directly measures alone-IPC during such high priority periods, rather than estimating alone request service rate and employs this alone-IPC estimate towards determining slowdown.

Applications of the MISE Model. The MISE model has been applied towards slowdown estimation in multiple contexts. Zhou and Wentzlaff [120] employ the MISE model in the context of throttling memory traffic at the source, based on inter-arrival times between requests. Specifically, they employ a set of bins, each corresponding to a range of inter-arrival times, and allocate a certain number of credits to each bin, depending on an application’s request inter-arrival times. In order to determine the optimal credit allocation in different bins corresponding to different arrival times, they employ a genetic algorithm. This credit allocation determines the eventual number of requests that can be served corresponding to different inter-arrival times, for an application, and hence,

shapes the memory traffic of the application. Slowdown estimates from the MISE model are leveraged to determine the optimal bins/credits configuration, to effectively shape memory traffic. Camouflage [119] employs the MISE model for the purposes of traffic shaping, but in the context of providing security. Camouflage shapes memory traffic into a predetermined distribution, in order to prevent attackers from probing the memory bus to infer the program’s memory access and response patterns. Slowdown estimates from the MISE model are used to determine the optimal bins/credits configuration.

Employing Slowdown-Proportional Resource Allocation. The general principle of allocating resources proportionally, to the estimated slowdown at that resource is a key principle employed in the MISE-QoS and MISE-Fair schemes. Two prior works [66, 108] apply a similar principle in the context of addressing interference at the on-chip network. Towards mitigating on-chip network contention, they build a scheme that allocates channel bandwidth proportional to the aggregate rate of flow of traffic from each thread.

These works [66, 98, 110, 119, 120] are clear instances of the applicability of the MISE model itself and its principles in various contexts. The works that build on our original MISE paper [97] are strongly indicative of the potential impact this work could have in the long term, as we describe in the next section.

6.2. Long-Term Impact

Predictable Performance in Current and Future Systems. Building predictable systems is a grand research challenge [12, 75, 78]. Predictable performance is a key requirement in current and future systems where 1) multiple applications are consolidated onto the same machine, sharing resources *and* 2) some applications need a certain guaranteed performance. Data centers, virtualized systems, interactive mobile systems and real-time systems are all examples of scenarios where predictable performance is desirable or necessary. We expect the need for predictable performance to increase in the future as more systems will likely move towards consolidation as a means to effectively utilize resources. Given this trend, accurately quantifying the effect of shared resource interference on performance is an important enabler towards providing predictable performance. Therefore, we believe that slowdown estimates from the MISE model and the hardware/software techniques that can be built on top of our model are important steps towards providing predictable performance.

Request Service Rate a Proxy for Performance. One of the key ideas behind MISE is to use memory request service rate as a proxy for performance for memory-bound applications. We hypothesize that the performance of an application that is bottlenecked at a certain resource is likely correlated with the request service rate at that resource. Hence, the notion of using request service rate as a proxy for performance can be used as a primitive for performance prediction

and applied more generally to other shared resources such as shared caches, storage and network. ASM [98], described in Section 6.1, is one such work that takes advantage of this key idea of request service rate as a proxy for performance, measured at the shared caches.

Accurate and Efficient Estimation of Alone Performance. Another key idea behind MISE is to periodically give each application the highest priority in order to estimate *alone-request-service-rate*. In doing so, the highest priority application receives minimal interference when its slowdown is being estimated, while also not disrupting other applications' execution. This leads to better accuracy than previous work [15, 17, 19, 76] that estimates an application's slowdown *while* it is receiving interference from other applications. We believe that the principle of estimating slowdown while using techniques such as prioritization to minimize interference can be applied at other shared resources such as I/O, storage and network as well.

Enabling Better Resource Management. The ability to accurately estimate slowdown in the presence of shared resource interference can enable a range of resource management techniques to provide QoS in both hardware and software. Slowdown estimates can be leveraged in the hardware for resource management (as we demonstrate with memory bandwidth). Slowdown estimates can also be communicated to the software, enabling more effective and informed admission control and migration mechanisms across a cluster of machines. Therefore, we believe MISE's slowdown estimates can enable substantial future research on resource allocation policies.

Simplicity of the Technique. The MISE model requires only simple hardware changes to the memory controller and scheduling logic, while providing high accuracy. By virtue of the memory bandwidth partitioning scheme we employ, the memory scheduler only needs to give one application the highest priority at any point in time, while treating other applications' requests similarly. On the other hand, previously proposed memory scheduling policies such as ATLAS, TCM [42, 43] employ ranking policies where an ordered ranking is enforced across all applications' requests. Hence, MISE requires simpler comparator logic compared to previous proposals and can be more easily incorporated into today's memory controllers than previous proposals.

Applicability to Other Memory Technologies. In our HPCA 2013 paper [97], we described MISE within the context of a system using DRAM as main memory, for which the reader can find detailed background information in our prior works [6, 8, 9, 10, 28, 29, 40, 41, 42, 43, 44, 45, 54, 55, 56, 57, 58, 62, 63, 83, 93, 94]. We believe the principles of MISE are easily applicable to other memory technologies, e.g., phase-change memory [47, 48, 49, 87, 107, 112, 118], STT-MRAM [46, 70, 79], and hybrid memory systems [1, 4, 11, 16, 26, 59, 65, 70, 71, 84, 86, 87, 88, 113, 114, 116]. We leave a detailed exploration of these to future works.

7. Conclusion

Application slowdowns induced by memory interference are a significant deterrent to high and predictable performance. Towards tackling such application slowdowns, our HPCA 2013 paper [97] (1) builds a simple Memory Interference-induced Slowdown Estimation (MISE) model to accurately estimate application slowdowns, and (2) demonstrates two use cases that leverage our MISE model to achieve predictable performance and fairness. Since our original HPCA 2013 paper [97] on the MISE model and its applications, several works have adopted and employed the MISE model and its principles in different contexts. We conclude that the MISE model and the principles behind it can fuel and inspire many more such works on high performance, predictable, and fair memory systems.

Acknowledgments

We thank Saugata Ghose for his dedicated effort in the preparation of this article. We thank the reviewers for their valuable feedback and suggestions. We acknowledge members of the SAFARI group for their feedback and for the stimulating research environment they provide. Many thanks to Brian Prasky from IBM and Arup Chakraborty from Freescale for their helpful comments. We acknowledge the support of our industrial sponsors, including AMD, HP Labs, IBM, Intel, Oracle, Qualcomm and Samsung. This research was also partially supported by the NSF (grant 0953246), SRC, and Intel URO Memory Hierarchy Program.

References

- [1] N. Agarwal and T. F. Wenisch, "Thermostat: Application-Transparent Page Management for Two-Tiered Main Memory," in *ASPLOS*, 2017.
- [2] R. Ausavarungrun et al., "Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems," in *ISCA*, 2012.
- [3] E. Baydal et al., "A Family of Mechanisms for Congestion Control in Wormhole Networks," *IEEE TPDS*, 2005.
- [4] S. Bock, B. R. Childers, R. Melhem, and D. Mossé, "Concurrent Migration of Multiple Pages in Software-Managed Hybrid Main Memory," in *ICCD*, 2016.
- [5] F. J. Cazorla et al., "Predictable performance in SMT processors: Synergy between the OS and SMTs," *IEEE TC*, Jul. 2006.
- [6] K. K. Chang, D. Lee, Z. Chishtii, A. R. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu, "Improving DRAM Performance by Parallelizing Refreshes with Accesses," in *HPCA*, 2014.
- [7] K. K. Chang et al., "HAT: Heterogeneous adaptive throttling for on-chip networks," in *SBAC-PAD '12*, 2012.
- [8] K. K. Chang, A. Kashyap, H. Hassan, S. Ghose, K. Hsieh, D. Lee, T. Li, G. Pekhimenko, S. Khan, and O. Mutlu, "Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization," in *SIGMETRICS*, 2016.
- [9] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu, "Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM," in *HPCA*, 2016.
- [10] K. K. Chang, A. G. Yağlıkçı, S. Ghose, A. Agrawal, N. Chatterjee, A. Kashyap, D. Lee, M. O'Connor, H. Hassan, and O. Mutlu, "Understanding Reduced-Voltage Operation in Modern DRAM Devices: Experimental Characterization, Analysis, and Mechanisms," in *SIGMETRICS*, 2017.
- [11] N. Chatterjee et al., "Leveraging Heterogeneity in DRAM Main Memories to Accelerate Critical Word Access," in *MICRO*, 2012.
- [12] Computing Research Association, "Grand research challenges in information systems," 2003.
- [13] R. Das et al., "Application-aware prioritization mechanisms for on-chip networks," in *MICRO*, 2009.
- [14] R. Das et al., "Application-to-Core Mapping Policies to Reduce Memory System Interference in Multi-Core Systems," in *HPCA*, 2013.
- [15] K. Du Bois et al., "Per-thread cycle accounting in multicore processors," in *HiPEAC*, 2013.

- [16] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan, "Data Tiering in Heterogeneous Memory Systems," in *EuroSys*, 2016.
- [17] E. Ebrahimi *et al.*, "Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems," in *ASPLOS*, 2010.
- [18] E. Ebrahimi *et al.*, "Parallel application memory scheduling," in *MICRO*, 2011.
- [19] E. Ebrahimi *et al.*, "Prefetch-aware shared resource management for multi-core systems," in *ISCA*, 2011.
- [20] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten, "Cache Pirating: Measuring the Curse of the Shared Cache," in *ICPP*, 2011.
- [21] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten, "Bandwidth Bandit: Quantitative Characterization of Memory Contention," in *PACT*, 2012.
- [22] S. Eyerma and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, 2008.
- [23] S. Eyerma and L. Eeckhout, "Per-thread cycle accounting in SMT processors," in *ASPLOS*, 2009.
- [24] S. Eyerma *et al.*, "A performance counter architecture for computing accurate CPI components," in *ASPLOS*, 2006.
- [25] S. Eyerma *et al.*, "A mechanistic performance model for superscalar out-of-order processors," *TOCS*, May 2009.
- [26] K. Gai, M. Qiu, H. Zhao, and L. Qiu, "Smart Energy-Aware Data Allocation for Heterogeneous Memory," in *HPCC*, 2016.
- [27] S. Ghose, H. Lee, and J. F. Martinez, "Improving Memory Scheduling via Processor-Side Load Criticality Information," in *ISCA*, 2013.
- [28] H. Hassan, N. Vijaykumar, S. Khan, S. Ghose, K. Chang, G. Pekhimenko, D. Lee, O. Ergin, and O. Mutlu, "SoftMC: A Flexible and Practical Open-Source Infrastructure for Enabling Experimental DRAM Studies," in *HPCA*, 2017.
- [29] H. Hassan, G. Pekhimenko, N. Vijaykumar, V. Seshadri, D. Lee, O. Ergin, and O. Mutlu, "ChargeCache: Reducing DRAM Latency by Exploiting Row Access Locality," in *HPCA*, 2016.
- [30] A. Herdrich *et al.*, "Rate-based QoS techniques for cache/memory in CMP platforms," in *ICS*, 2009.
- [31] Intel Corp., "First the tick, now the tock: Next generation Intel microarchitecture (Nehalem)," White Paper, 2008.
- [32] E. Ipek *et al.*, "Self-optimizing memory controllers: A reinforcement learning approach," in *ISCA*, 2008.
- [33] R. Iyer, "CQoS: A framework for enabling QoS in shared caches of CMP platforms," in *ICS*, 2004.
- [34] R. Iyer *et al.*, "QoS policies and architecture for cache/memory in CMP platforms," in *SIGMETRICS*, 2007.
- [35] M. Jahre and L. Eeckhout, "GDP: Using Dataflow Properties to Accurately Estimate Interference-free Performance at Runtime," in *HPCA*, 2018.
- [36] M. K. Jeong, D. H. Yoon, D. Sunwoo, M. Sullivan, I. Lee, and M. Erez, "Balancing DRAM locality and parallelism in shared memory CMP systems," in *HPCA*, 2012.
- [37] T. S. Karkhanis and J. E. Smith, "A first-order superscalar processor model," in *ISCA*, 2004.
- [38] D. Kaseridis *et al.*, "Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era," in *MICRO*, 2011.
- [39] O. Kayiran, N. C. Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das, "Managing GPU Concurrency in Heterogeneous Architectures," in *MICRO*, 2014.
- [40] J. S. Kim, M. Patel, H. Hassan, and O. Mutlu, "The DRAM Latency PUF: Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency-Reliability Tradeoff in Modern DRAM Devices," in *HPCA*, 2018.
- [41] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," in *ISCA*, 2014.
- [42] Y. Kim *et al.*, "ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *HPCA*, 2010.
- [43] Y. Kim *et al.*, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *MICRO*, 2010.
- [44] Y. Kim *et al.*, "A case for exploiting subarray-level parallelism (salp) in dram," in *ISCA*, 2012.
- [45] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," *CAL*, 2015.
- [46] E. Kültürsay, M. Kandemir, A. Sivasubramanian, and O. Mutlu, "Evaluating STT-RAM as an Energy-Efficient Main Memory Alternative," in *ISPASS*, 2013.
- [47] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting Phase Change Memory as a Scalable DRAM Alternative," in *ISCA*, 2009.
- [48] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Phase Change Memory Architecture and the Quest for Scalability," *CACM*, 2010.
- [49] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-Change Technology and the Future of Main Memory," *IEEE Micro*, 2010.
- [50] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt, "Prefetch-Aware DRAM Controllers," in *MICRO*, 2008.
- [51] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt, "Prefetch-Aware Memory Controllers," *TC*, 2011.
- [52] 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," Univ. of Texas at Austin, High Performance Systems Group, Tech. Rep. TR-HPS-2010-002, 2010.
- [53] C. J. Lee, V. Narasiman, O. Mutlu, and Y. N. Patt, "Improving Memory Bank-Level Parallelism in the Presence of Prefetching," in *MICRO*, 2009.
- [54] D. Lee, S. Khan, L. Subramanian, S. Ghose, R. Ausavarungnirun, G. Pekhimenko, V. Seshadri, and O. Mutlu, "Design-Induced Latency Variation in Modern DRAM Chips: Characterization, Analysis, and Latency Reduction Mechanisms," in *SIGMETRICS*, 2017.
- [55] D. Lee, S. Ghose, G. Pekhimenko, S. Khan, and O. Mutlu, "Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost," *TACO*, 2016.
- [56] D. Lee, Y. Kim, G. Pekhimenko, S. Khan, V. Seshadri, K. Chang, and O. Mutlu, "Adaptive-Latency DRAM: Optimizing DRAM Timing for the Common-Case," in *HPCA*, 2015.
- [57] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu, "Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture," in *HPCA*, 2013.
- [58] D. Lee, L. Subramanian, R. Ausavarungnirun, J. Choi, and O. Mutlu, "Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM," in *PACT*, 2015.
- [59] Y. Li, S. Ghose, J. Choi, J. Sun, H. Wang, and O. Mutlu, "Utility-Based Hybrid Memory Management," in *CLUSTER*, 2017.
- [60] X. Lin and R. Balasubramanian, "Refining the Utility Metric for Utility-Based Cache Partitioning," in *WDDD*, 2009.
- [61] F. Liu, X. Jiang, and Y. Solihin, "Understanding How Off-Chip Memory Bandwidth Partitioning in Chip Multiprocessors Affects System Performance," in *HPCA*, 2010.
- [62] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu, "An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms," in *ISCA*, 2013.
- [63] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, "RAIDR: Retention-Aware Intelligent DRAM Refresh," in *ISCA*, 2012.
- [64] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu, "A Software Memory Partitioning Approach for Eliminating Bank-level Interference in Multicore Systems," in *PACT*, 2012.
- [65] L. Liu, H. Yang, Y. Li, M. Xie, L. Li, and C. Wu, "Memos: A Full Hierarchy Hybrid Memory Management Framework," in *ICCD*, 2016.
- [66] Z. Lu *et al.*, "Aggregate flow-based performance fairness in CMPs," *TACO*, vol. 13, no. 4, Dec. 2016.
- [67] K. Luo *et al.*, "Balancing throughput and fairness in SMT processors," in *ISPASS*, 2001.
- [68] C. Luque *et al.*, "CPU accounting in CMP processors," *IEEE CAL*, Jan. - Jun. 2009.
- [69] J. Mars *et al.*, "Bubble-Up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *MICRO*, 2011.
- [70] J. Meza, Y. Luo, S. Khan, J. Zhao, Y. Xie, and O. Mutlu, "A Case for Efficient Hardware/Software Cooperative Management of Storage and Memory," in *WEED*, 2013.
- [71] J. Meza *et al.*, "Enabling efficient and scalable hybrid memories using fine-granularity DRAM cache management," *CAL*, 2012.
- [72] T. Moscibroda and O. Mutlu, "Memory performance attacks: Denial of memory service in multi-core systems," in *USENIX Security*, 2007.
- [73] T. Moscibroda and O. Mutlu, "Distributed Order Scheduling and its Application to Multi-Core DRAM Controllers," in *PODC*, 2008.
- [74] S. P. Muralidhara *et al.*, "Reducing memory interference in multicore systems via application-aware memory channel partitioning," in *MICRO*, 2011.
- [75] O. Mutlu, "Memory Scaling: A Systems Architecture Perspective," *IMW*, 2013.
- [76] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *MICRO*, 2007.
- [77] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems," in *ISCA*, 2008.
- [78] O. Mutlu and L. Subramanian, "Research Problems and Opportunities in Memory Systems," *SUPERFRI*, 2014.
- [79] H. Naeimi, C. Augustine, A. Raychowdhury, S.-L. Lu, and J. Tschanz, "STT-RAM Scaling and Retention Failure," *Intel Technology Journal*, 2013.
- [80] K. J. Nesbit *et al.*, "Fair queuing memory systems," in *MICRO*, 2006.
- [81] G. Nychis, C. Fallin, T. Moscibroda, and O. Mutlu, "Next Generation On-Chip Networks: What Kind of Congestion Control Do We Need?" in *HotNets*, 2010.
- [82] G. Nychis, C. Fallin, T. Moscibroda, and O. Mutlu, "On-Chip Networks from a Networking Perspective: Congestion and Scalability in Many-core Interconnects," in *SIGCOMM*, 2012.
- [83] M. Patel, J. S. Kim, and O. Mutlu, "The Reach Profiler (REAPER): Enabling the Mitigation of DRAM Retention Failures via Profiling at Aggressive Conditions," in *ISCA*, 2017.
- [84] A. J. Peña and P. Balaji, "Toward the Efficient Use of Multiple Explicitly Managed Memory Subsystems," in *CLUSTER*, 2014.
- [85] D. Petrou *et al.*, "Implementing lottery scheduling: Matching the specializations in traditional schedulers," in *USENIX ATEC*, 1999.
- [86] S. Phadke *et al.*, "MLP Aware Heterogeneous Memory System," in *DATE*, 2011.
- [87] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable High Performance Main Memory System Using Phase-Change Memory Technology," in *ISCA*, 2009.
- [88] L. E. Ramos, E. Gorbato, and R. Bianchini, "Page Placement in Hybrid Memory Systems," in *ICS*, 2011.
- [89] S. Rixner *et al.*, "Memory access scheduling," in *ISCA*, 2000.

- [90] SAFARI Research Group, *ASMSim – GitHub Repository*, <https://github.com/CMU-SAFARI/ASMSim>.
- [91] A. Sandberg, A. Sembrant, E. Hagersten, and D. Black-Schaffer, "Modeling Performance Variation Due to Cache Sharing," in *HPCA*, 2013.
- [92] V. Seshadri *et al.*, "The evicted-address filter: A unified mechanism to address both cache pollution and thrashing," in *PACT*, 2012.
- [93] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology," in *MICRO*, 2017.
- [94] V. Seshadri *et al.*, "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization," in *MICRO*, 2013.
- [95] A. Snively and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multi-threaded processor," in *ASPLOS*, 2000.
- [96] Standard Performance Evaluation Corp., *SPEC CPU2006*, <http://www.spec.org/spec2006>.
- [97] L. Subramanian *et al.*, "MISE: Providing performance predictability and improving fairness in shared main memory systems," in *HPCA*, 2013.
- [98] L. Subramanian *et al.*, "The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory," in *MICRO*, 2015.
- [99] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, "The blacklisting memory scheduler: Achieving high performance and fairness at low cost," in *ICCD*, 2014.
- [100] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, "BLISS: Balancing performance, fairness and complexity in memory access scheduling," *TPDS*, 2016.
- [101] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, "The Impact of Memory Subsystem Resource Sharing on Datacenter Applications," in *ISCA*, 2011.
- [102] M. Thottethodi, A. R. Lebeck, and S. Mukherjee, "Self-Tuned Congestion Control for Multiprocessor Networks," in *HPCA*, 2001.
- [103] H. Usui *et al.*, "DASH: Deadline-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators," *TACO*, Jan. 2016.
- [104] K. Van Craeynest *et al.*, "Scheduling heterogeneous multi-cores through performance impact estimation (PIE)," in *ISCA*, 2012.
- [105] C. A. Waldspurger and W. E. Weihl, "Lottery scheduling: Flexible proportional-share resource management," in *OSDI*, 1994.
- [106] H. Wang *et al.*, "A-DRM: Architecture-aware distributed resource management of virtualized clusters," in *VEE*, 2015.
- [107] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase Change Memory," *Proc. IEEE*, 2010.
- [108] X. Xiang, W. Shi, S. Ghose, L. Peng, O. Mutlu, and N.-F. Tzeng, "Carpool: A Bufferless On-Chip Network Supporting Adaptive Multicast and Hotspot Alleviation," in *ICS*, 2017.
- [109] M. Xie, D. Tong, K. Huang, and X. Cheng, "Improving System Throughput and Fairness Simultaneously in Shared Memory CMP Systems via Dynamic Bank Partitioning," in *HPCA*, 2014.
- [110] D. Xiong *et al.*, "Providing predictable performance via a slowdown estimation model," *TACO*, vol. 14, no. 3, Aug. 2017.
- [111] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers," in *ISCA*, 2013.
- [112] H. Yoon, J. Meza, N. Muralimanohar, N. P. Jouppi, and O. Mutlu, "Efficient Data Mapping and Buffering Techniques for Multi-Level Cell Phase-Change Memories," *TACO*, 2014.
- [113] H. Yoon *et al.*, "Row Buffer Locality Aware Caching Policies for Hybrid Memories," in *ICCD*, 2012.
- [114] X. Yu, C. J. Hughes, N. Satish, O. Mutlu, and S. Devadas, "Banshee: Bandwidth-Efficient DRAM Caching via Software/Hardware Cooperation," in *MICRO*, 2017.
- [115] G. L. Yuan *et al.*, "Complexity effective memory access scheduling for many-core accelerator architectures," in *MICRO*, 2009.
- [116] W. Zhang and T. Li, "Exploring Phase Change Memory and 3D Die-Stacking for Power/Thermal Friendly, Fast and Durable Memory Architectures," in *PACT*, 2009.
- [117] J. Zhao, O. Mutlu, and Y. Xie, "FIRM: Fair and High-Performance Memory Control for Persistent Memory Systems," in *MICRO*, 2014.
- [118] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology," in *ISCA*, 2009.
- [119] Y. Zhou *et al.*, "Camouflage: Memory traffic shaping to mitigate timing attacks," in *HPCA*, 2017.
- [120] Y. Zhou and D. Wentzlaff, "MITTS: Memory inter-arrival time traffic shaping," in *ISCA*, 2016.
- [121] S. Zhuravlev *et al.*, "Addressing shared resource contention in multicore processors via scheduling," in *ASPLOS*, 2010.
- [122] W. K. Zuravleff and T. Robinson, "Controller for a synchronous DRAM that maximizes throughput by allowing memory requests and commands to be issued out of order," Patent 5630096, 1997.

High-Performance and Energy-Efficient Memory Scheduler Design for Heterogeneous Systems

Rachata Ausavarungnirun¹ Gabriel H. Loh²
Lavanya Subramanian^{3,1} Kevin Chang^{4,1} Onur Mutlu^{5,1}

¹Carnegie Mellon University ²AMD Research ³Intel Labs ⁴Facebook ⁵ETH Zürich

This paper summarizes the idea of the Staged Memory Scheduler (SMS), which was published at ISCA 2012 [14], and examines the work’s significance and future potential. When multiple processor cores (CPUs) and a GPU integrated together on the same chip share the off-chip DRAM, requests from the GPU can heavily interfere with requests from the CPUs, leading to low system performance and starvation of cores. Unfortunately, state-of-the-art memory scheduling algorithms are ineffective at solving this problem due to the very large amount of GPU memory traffic, unless a very large and costly request buffer is employed to provide these algorithms with enough visibility across the global request stream.

Previously-proposed memory controller (MC) designs use a single monolithic structure to perform three main tasks. First, the MC attempts to schedule together requests to the same DRAM row to increase row buffer hit rates. Second, the MC arbitrates among the requesters (CPUs and GPU) to optimize for overall system throughput, average response time, fairness and quality of service. Third, the MC manages the low-level DRAM command scheduling to complete requests while ensuring compliance with all DRAM timing and power constraints.

This paper proposes a fundamentally new approach, called the Staged Memory Scheduler (SMS), which decouples the three primary MC tasks into three significantly simpler structures that together improve system performance and fairness. Our three-stage MC first groups requests based on row buffer locality. This grouping allows the second stage to focus only on inter-application scheduling decisions. These two stages enforce high-level policies regarding performance and fairness, and therefore the last stage can use simple per-bank FIFO queues (i.e., there is no need for further command reordering within each bank) and straightforward logic that deals only with the low-level DRAM commands and timing.

We evaluated the design trade-offs involved and compared it against four state-of-the-art MC designs. Our evaluation shows that SMS provides 41.2% performance improvement and 4.8× fairness improvement compared to the best previous state-of-the-art technique, while enabling a design that is significantly less complex and more power-efficient to implement.

Our analysis and proposed scheduler have inspired significant research on (1) predictable and/or deadline-aware memory scheduling [91, 92, 194, 195, 197, 201, 202, 216] and (2) memory scheduling for heterogeneous systems [161, 201, 202, 207].

1. Introduction

As the number of cores continues to increase in modern chip multiprocessor (CMP) systems, the DRAM memory system has become a critical shared resource [139, 145]. Memory requests from multiple cores interfere with each other, and this inter-application interference is a significant impediment to individual application and overall system performance. Various works on application-aware memory scheduling [98, 99, 141, 142] address the problem by making the memory controller aware of application characteristics and appropriately prioritizing memory requests to improve system performance and fairness.

Recent heterogeneous CPU-GPU systems [1, 27, 28, 37, 76, 77, 78, 133, 152, 153, 167, 209] present an additional challenge by introducing integrated graphics processing units (GPUs) on the same die with CPU cores. GPU applications typically demand significantly more memory bandwidth than CPU applications due to the GPU’s capability of executing a large number of concurrent threads [1, 2, 3, 4, 13, 23, 27, 37, 38, 40, 62, 68, 77, 78, 133, 149, 150, 151, 152, 153, 154, 167, 176, 178, 179, 188, 189, 199, 200, 206, 209]. GPUs use *single-instruction multiple-data* (SIMD) pipelines to concurrently execute multiple threads [53]. In a GPU, a group of threads executing the same instruction is called a *wavefront* or *warp*, and threads in a warp are executed in lockstep. When a wavefront stalls on a memory instruction, the GPU core hides this memory access latency by switching to another wavefront to avoid stalling the pipeline. Therefore, there can be thousands of outstanding memory requests from across all of the wavefronts. This is fundamentally more memory intensive than CPU memory traffic, where each CPU application has a much smaller number of outstanding requests due to the sequential execution model of CPUs.

Figure 1 (a) shows the memory request rates for a representative subset of our GPU applications and the most memory-intensive SPEC2006 (CPU) applications, as measured by memory requests per thousand cycles when each application runs alone on the system. The raw bandwidth demands (i.e., memory request rates) of the GPU applications are often multiple times higher than the SPEC benchmarks. Figure 1 (b) shows the row buffer hit rates (also called *row buffer locality* or RBL [134]). The GPU applications show consistently high levels of RBL, whereas the SPEC benchmarks exhibit more variability. The GPU programs have high levels of spatial

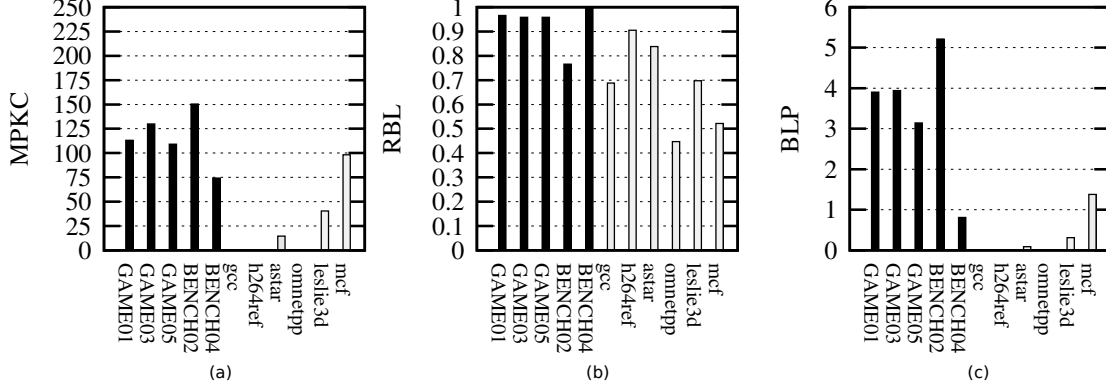


Figure 1: GPU memory characteristics. (a) Memory intensity, measured by memory requests per thousand cycles, (b) row buffer locality, measured by the fraction of accesses that hit in the row buffer, and (c) bank-level parallelism. Reproduced from [14].

locality, often due to access patterns related to large sequential memory accesses (e.g., frame buffer updates). Figure 1(c) shows the *bank-level parallelism (BLP)* [109, 142], which is the average number of parallel memory requests that can be issued to different DRAM banks, for each application, with the GPU programs consistently making use of *four* banks at the same time.

In addition to the high-intensity memory traffic of GPU applications, there are other properties that distinguish GPU applications from CPU applications. Prior work [99] observed that CPU applications with streaming access patterns typically exhibit high RBL but low BLP, while applications with less uniform access patterns typically have low RBL but high BLP.

In contrast, GPU applications have *both* high RBL and high BLP. The combination of high memory intensity, high RBL and high BLP means that the GPU will cause significant interference to other applications across all banks, especially when using a memory scheduling algorithm that preferentially favors requests that result in row buffer hits (e.g., [173, 220]).

Recent memory scheduling research has focused on memory interference between applications in CPU-only scenarios. These past proposals are built around a single centralized request buffer at each memory controller (MC). The scheduling algorithm implemented in the memory controller analyzes the stream of requests in the centralized request buffer to determine application memory characteristics, decides on a priority for each core, and then enforces these priorities. Observable memory characteristics may include the number of requests that result in row buffer hits, the bank-level parallelism of each core, memory request rates, overall fairness metrics, and other information. Figure 2(a) shows the CPU-only scenario where the request buffer only holds requests from the CPUs. In this case, the memory controller sees a number of requests from the CPUs and has visibility into their memory behavior. On the other hand, when the request buffer is shared between the CPUs and the GPU, as shown in Figure 2(b), the large volume of requests from the GPU occupies a significant fraction of the memory controller’s request



Figure 2: Example of the limited visibility of the memory controller. (a) CPU-only information, (b) Memory controller’s visibility, (c) Improved visibility. Adapted from [14].

buffer, thereby limiting the memory controller’s visibility of the CPU applications’ memory characteristics.

One approach to increasing the memory controller’s visibility across a larger window of memory requests is to increase the size of its request buffer. This allows the memory controller to observe more requests from the CPUs to better characterize their memory behavior, as shown in Figure 2(c). For instance, with a large request buffer, the memory controller can identify and service multiple requests from one CPU core to the same row such that they become row buffer hits, however, with a small request buffer as shown in Figure 2(b), the memory controller may not even see these requests at the same time because the GPU’s requests have occupied the majority of the entries.

Unfortunately, very large request buffers impose significant implementation challenges, including the die area for the larger structures and the additional circuit complexity for analyzing so many requests, along with the logic needed for assignment and enforcement of priorities [194, 195]. Therefore, while building a very large, centralized memory controller request buffer could perhaps lead to reasonable memory scheduling decisions, the approach is unattractive due to the resulting area, power, timing and complexity costs.

In this work, we propose the Staged Memory Scheduler (SMS), a decentralized architecture for memory scheduling in the context of integrated multi-core CPU-GPU systems. The key idea in SMS is to decouple the various functional tasks of memory controllers and partition these tasks *across* several simpler hardware structures which operate in a sta-

ged fashion. The three primary functions of the memory controller, which map to the three stages of our proposed memory controller architecture, are:

1. Detection of basic intra-application memory characteristics (e.g., row buffer locality).
2. Prioritization across applications (CPUs and GPU) and enforcement of policies to reflect the priorities.
3. Low-level command scheduling (e.g., activate, precharge, read/write), enforcement of DRAM device timing constraints (e.g., t_{RAS} , t_{FAW} , etc.), and resolution of resource conflicts (e.g., data bus arbitration).¹

Our specific SMS implementation makes widespread use of distributed FIFO structures to maintain a very simple implementation, but at the same time SMS can provide fast service to low memory-intensity (likely latency-sensitive) applications and effectively exploit row buffer locality and bank-level parallelism for high memory-intensity (bandwidth-demanding) applications. While SMS provides a specific implementation, our staged approach for memory controller organization provides a general framework for exploring scalable memory scheduling algorithms capable of handling the diverse memory needs of integrated heterogeneous processing systems of the future (e.g., systems-on-chip that contain CPUs, GPUs, and accelerators).

2. Staged Memory Scheduler Design

Overview: Our proposed Staged Memory Scheduler [14] architecture introduces a new memory controller (MC) design that provides 1) scalability and simpler implementation by decoupling the primary functions of an application-aware MC into a simpler multi-stage MC, and 2) performance and fairness improvement by reducing the interference caused by very bandwidth-intensive applications. SMS provides these benefits by introducing a three-stage design. The first stage is the per-core *batch formation* stage, which groups requests from the same application that access the same row to improve row buffer locality. The second stage is the *batch scheduler*, which schedules batches of requests from across different applications. The last stage is the *DRAM command scheduler*, which sends requests to DRAM while satisfying all DRAM constraints.

The staged organization of SMS lends directly to a low-complexity hardware implementation. Figure 3 illustrates the overall hardware organization of the SMS. We briefly discuss each stage below. Section 4 of our ISCA 2012 paper [14] includes a detailed description of each stage.

Stage 1 - Batch Formation. The goal of this stage is to combine individual memory requests from each source into batches of requests that are to the same row buffer entry. It consists of several simple FIFO structures, one per *source* (i.e., a CPU core or the GPU). Each request from a given source

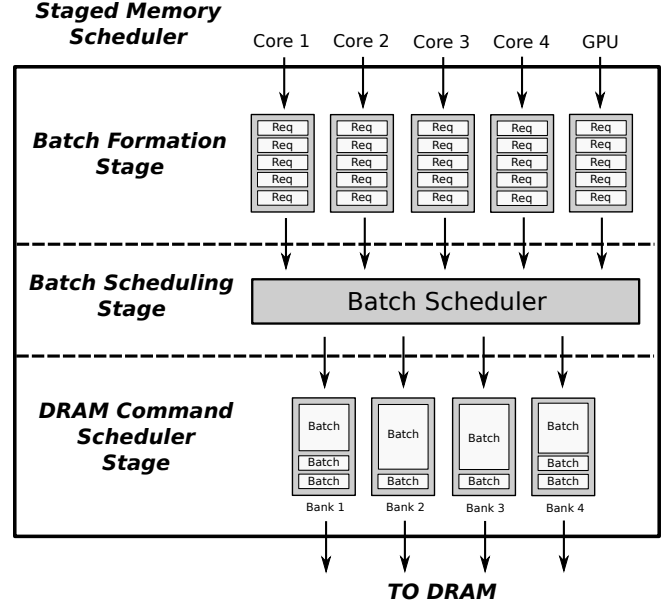


Figure 3: The organization of SMS. Adapted from [14].

is initially inserted into its respective FIFO upon arrival at the MC. A *batch* is simply one or more memory requests from the same source that access the same DRAM row. That is, all requests within a batch, except perhaps for the first one, would be row buffer hits if scheduled consecutively. A batch is deemed complete or *ready* when an incoming request accesses a different row, when the oldest request in the batch has exceeded a threshold age, or when the FIFO is full. Only ready batches are considered for future scheduling by the second stage of SMS.

Stage 2 - Batch Scheduler. The batch scheduler deals directly with batches, and therefore does not need to worry about optimizing for row buffer locality. Instead, the batch scheduler focuses on higher-level policies regarding inter-application interference and fairness. The goal of this stage is to prioritize batches from applications that are latency critical, while making sure that bandwidth-intensive applications (e.g., those running on the GPU) still make good progress.

The batch scheduler considers every source FIFO (from stage 1) that contains a ready batch. It picks one ready batch based on either a shortest job first (SJF) or a round-robin policy. Using the SJF policy, the batch scheduler chooses the oldest ready batch from the source with the fewest total in-flight memory requests across all three stages of SMS. SJF prioritization reduces average request service latency, and it tends to favor latency-sensitive applications, which tend to have fewer total requests [98, 99, 109, 142]. Using the round-robin policy, the batch scheduler simply picks the next ready batch in a round-robin manner across the source FIFOs. This ensures that memory-intensive applications receive adequate service. The batch scheduler uses the SJF policy with probability p and the round-robin policy with probability $1 - p$. The value of p determines whether the CPU or the GPU

¹We refer the reader to our prior works [32, 33, 34, 35, 66, 67, 93, 96, 97, 98, 99, 100, 112, 113, 114, 115, 116, 120, 121, 158, 183, 184] for a detailed background on DRAM.

receives higher priority. When p is high, the SJF policy is applied more often and applications with fewer outstanding requests are prioritized. Hence, the batches of the likely less memory-intensive CPU applications are prioritized over the batches of the GPU application. On the other hand, when p is low, request batches are scheduled in a round-robin fashion more often. Hence, the memory-intensive GPU application's naturally-large request batches are likely scheduled more frequently, and the GPU is thus prioritized over the CPU.

After picking a batch, the batch scheduler enters a drain state where it forwards the requests from the selected batch to the final stage of the SMS. The batch scheduler dequeues one request per cycle until all requests from the batch have been removed from the selected FIFO.

Stage 3 - DRAM Command Scheduler (DCS). DCS consists of one FIFO queue per DRAM bank. The drain state of the batch scheduler places the memory requests directly into these FIFOs. Note that because batches are moved into DCS FIFOs one batch at a time, row buffer locality within a batch is preserved within a DCS FIFO. At this point, higher-level policy decisions have already been made by the batch scheduler. Therefore, the DCS simply issues low-level DRAM commands, ensuring DRAM protocol compliance.

In any given cycle, DCS considers only the requests at the *head* of each of the per-bank FIFOs. For each request, DCS determines whether that request can issue a command based on the request's current row buffer state (e.g., is the row buffer already open with the requested row?) and the current DRAM state (e.g., time elapsed since a row was opened in a bank, and data bus availability). If more than one request is eligible to issue a command in any given cycle, the DCS arbitrates between DRAM banks in a round-robin fashion.

3. Qualitative Comparison with Previous Scheduling Algorithms

In this section, we compare SMS qualitatively to previously proposed scheduling policies and analyze the basic differences between SMS and these policies. The fundamental difference between SMS and previously-proposed memory scheduling policies for CPU-only scenarios is that the latter are designed around a single, centralized request buffer which has poor scalability and complex scheduling logic, while SMS is built around a decentralized, scalable framework.

First-Ready FCFS (FR-FCFS). FR-FCFS [173, 220] is a commonly used scheduling policy in commodity DRAM systems. An FR-FCFS scheduler prioritizes requests that result in row buffer hits over row buffer misses and otherwise prioritizes older requests. Since FR-FCFS unfairly prioritizes applications with high row buffer locality to maximize DRAM throughput, prior works [42, 45, 98, 99, 134, 137, 141, 142, 194, 195] have observed that it has low system performance and high unfairness.

Parallelism-Aware Batch Scheduling (PAR-BS). PAR-BS [142, 143] aims to improve both fairness and system per-

formance. In order to prevent unfairness, it forms batches of outstanding memory requests and prioritizes the oldest batch, to avoid request starvation. To improve system throughput, it prioritizes applications with smaller number of outstanding memory requests within a batch. However, PAR-BS has two major shortcomings. First, batching could cause older GPU requests and requests of other memory-intensive CPU applications to be prioritized over latency-sensitive CPU applications. Second, as previous work [98] has also observed, PAR-BS does not take into account an application's long term memory-intensity characteristics when it assigns application priorities within a batch. This could cause memory-intensive applications' requests to be prioritized over latency-sensitive applications' requests within a batch, due to the application-agnostic nature of batching.

Adaptive Per-Thread Least-Attained-Served Memory Scheduling (ATLAS). ATLAS [98] aims to improve system performance by prioritizing requests of applications with lower attained memory service. This improves the performance of low memory-intensity applications as they tend to have low attained service. However, ATLAS has the disadvantage of not preserving fairness. Previous works [98, 99] have shown that simply prioritizing applications based on attained service leads to significant slowdown of memory-intensive applications.

Thread Cluster Memory Scheduling (TCM). TCM [99] is a state-of-the-art application-aware cluster memory scheduler providing both high system throughput and high fairness. It groups an application into either a latency-sensitive or a bandwidth-sensitive cluster based on the application memory intensity. In order to achieve high system throughput and low unfairness, TCM employs a different prioritization policy for each cluster. To improve system throughput, a fraction of total memory bandwidth is dedicated to the latency-sensitive cluster and applications within the cluster are then ranked based on memory intensity with the least memory-intensive application receiving the highest priority. On the other hand, TCM minimizes unfairness by periodically shuffling applications within the bandwidth-sensitive cluster to avoid starvation. This approach provides both high system performance and fairness in CPU-only systems. In an integrated CPU-GPU system, the GPU generates a significantly larger number of memory requests compared to the CPUs and fills up the centralized request buffer. As a result, the memory controller lacks the visibility into CPU memory requests to accurately determine each application's memory access characteristics. Without such visibility, TCM makes incorrect and non-robust clustering decisions, which classify some applications with high memory intensity into the latency-sensitive cluster and vice versa. Such misclassified applications cause interference not only to low memory intensity applications, but also to each other. Therefore, TCM cannot always provide high system performance and high fairness in an integrated CPU-GPU system. Increasing the

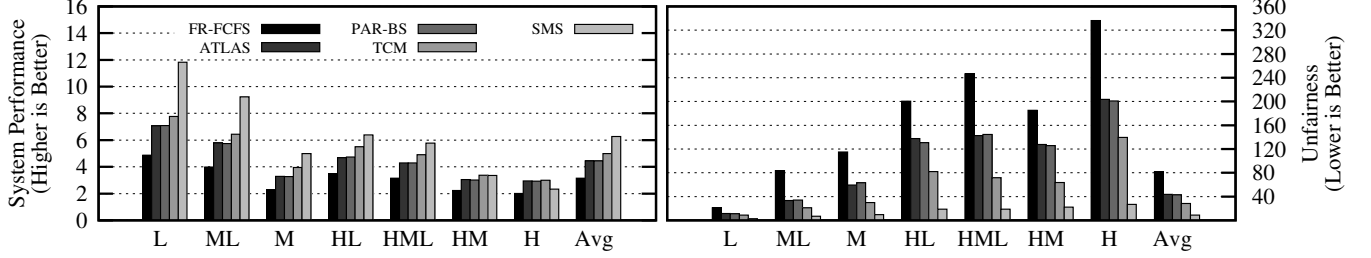


Figure 4: System performance, and fairness for 7 categories of workloads (total of 105 workloads). Reproduced from [14].

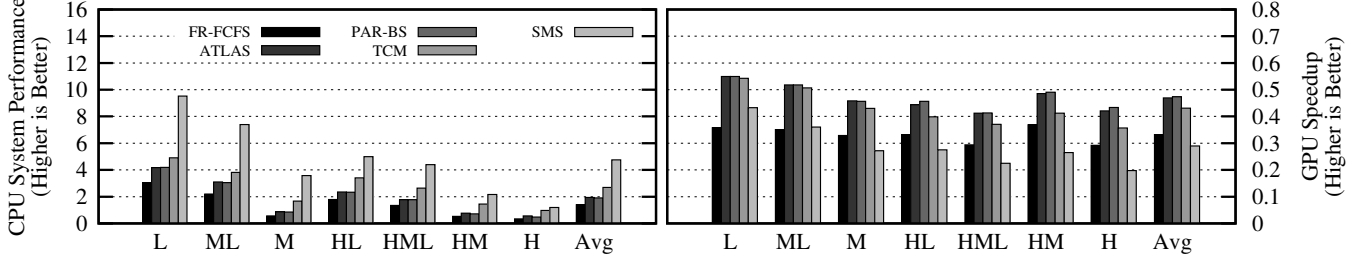


Figure 5: CPUs and GPU Speedup for 7 categories of workloads (total of 105 workloads). Reproduced from [14].

request buffer size is a practical way to gain more visibility into CPU applications’ memory access characteristics. However, this approach is not scalable as we show in our evaluations [14]. In contrast, SMS provides much better system performance and fairness than TCM with the same number of request buffer entries and lower hardware cost, as we show in Section 5.

4. Evaluation Methodology

We use an in-house cycle-accurate simulator to perform our evaluations. For our performance evaluations, we model a system with sixteen x86 CPU cores and a GPU. For the CPUs, we model three-wide out-of-order processors with a cache hierarchy including per-core L1 caches and a shared, distributed L2 cache. The GPU does not share the CPU caches. In order to prevent the GPU from taking the majority of request buffer entries, we reserve half of the request buffer entries for the CPUs. To model the memory bandwidth of the GPU accurately, we perform coalescing on GPU memory requests before they are sent to the memory controller [119].

We evaluate our system with a set of 105 multiprogrammed workloads simulated for 500 million cycles. Each workload consists of sixteen SPEC CPU2006 benchmarks and one GPU application selected from a mix of video games and graphics performance benchmarks. We classify CPU benchmarks into three categories (Low, Medium, and High) based on their memory intensities, measured as last-level cache misses per thousand instructions (MPKI). Based on these three categories, we randomly choose sixteen CPU benchmarks from these three categories and one randomly selected GPU benchmark to form workloads consisting of seven intensity mixes: L (All low), ML (Low/Medium), M (All medium), HL (High/Low), HML (High/Medium/Low), HM (High/Medium) and H (All high). For each CPU benchmark, we use Pin [125, 172] with

PinPoints [159] to select the representative phase. For the GPU applications, we use an industrial GPU simulator to collect memory requests with detailed timing information. These requests are collected after having first been filtered through the GPU’s internal cache hierarchy, therefore we do not further model any caches for the GPU in our final hybrid CPU-GPU simulation framework. More detail on our experimental methodology is in Section 5 of our ISCA 2012 paper [14].

5. Experimental Results

We present the performance of five memory scheduler configurations: FR-FCFS [173, 220], ATLAS [98], PAR-BS [142], TCM [99], and SMS [14] on the 16-CPU/1-GPU four-memory-controller system. All memory schedulers use 300 request buffer entries per memory controller. This size was chosen based on empirical results, which showed that performance does not appreciably increase for larger request buffer sizes. Results are presented in the workload categories, with workload memory intensities increasing from left to right.

Figure 4 shows the system performance (measured as weighted speedup [50, 51]) and fairness (measured as maximum slowdown [43, 98, 99, 194, 195, 203]) of the previously proposed algorithms and SMS. Compared to TCM, which is the previous state-of-the-art algorithm for both system performance and fairness, SMS provides 41.2% system performance improvement and $4.8\times$ fairness improvement. Therefore, we conclude that SMS provides better system performance and fairness than all previously proposed scheduling policies, while incurring much lower hardware cost and simpler scheduling logic, as we show in Section 5.2.

We study the performance of the CPU system and the GPU system separately and provide two major observations in Figure 5. First, SMS gains $1.76\times$ improvement in CPU

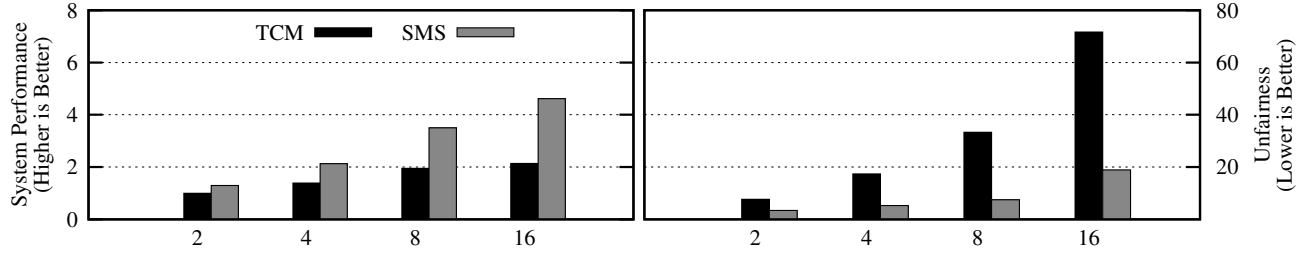


Figure 6: SMS vs. TCM on a 16 CPU/1 GPU, 4 memory controller system with varying the number of cores.

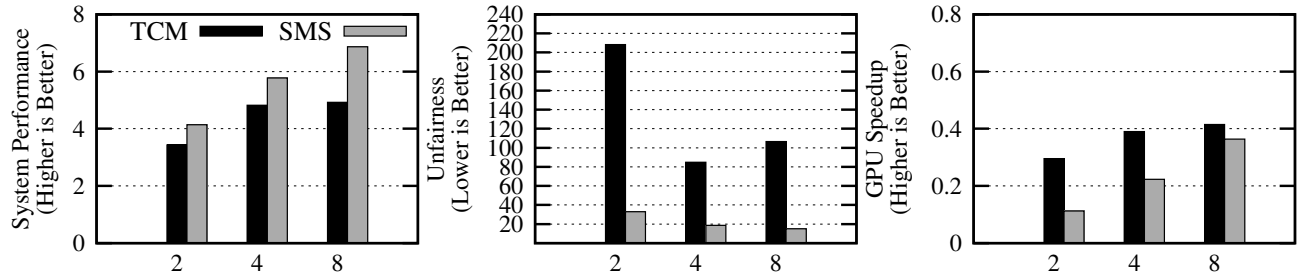


Figure 7: SMS vs. TCM on a 16 CPU/1 GPU system with varying the number of channels.

system performance over TCM. Second, SMS achieves this $1.76\times$ CPU performance improvement while delivering similar GPU performance as the FR-FCFS baseline. The results show that TCM (and the other algorithms) end up allocating far more bandwidth to the GPU, at significant performance and fairness cost to the CPU applications. SMS appropriately deprioritizes the memory bandwidth intensive GPU application in order to enable higher CPU performance and overall system performance, while preserving fairness. Previously proposed scheduling algorithms, on the other hand, allow the GPU to hog memory bandwidth and therefore significantly degrade system performance and fairness.

We provide a more detailed analysis in Sections 6.1 and 6.2 of our ISCA 2012 paper [14].

5.1. Scalability with Cores and Memory Controllers

Figure 6 compares the performance and fairness of SMS against TCM (averaged over 75 workloads)² with the same number of request buffers, as the number of cores is varied. We make the following observations: First, SMS continues to provide better system performance and fairness than TCM. Second, the system performance gains and fairness gains increase significantly as the number of cores and hence, memory pressure is increased. SMS’s performance and fairness benefits are likely to become more significant as core counts in future technology nodes increase.

Figure 7 shows the system performance and fairness of SMS compared against TCM as the number of memory channels is varied. For this, and all subsequent results, we perform our evaluations on 60 workloads from categories that contain

²We use 75 randomly selected workloads per core count. We could not use the same workloads/categorizations as specified in Section 4 because those were for 16-core systems, whereas we are now varying the number of cores.

high memory-intensity applications (HL, HML, HM and H workload categories). We observe that SMS scales better as the number of memory channels increases. As the performance gain of TCM diminishes when the number of memory channels increases from 4 to 8 channels, SMS continues to provide performance improvement for both CPU and GPU. We provide a detailed scalability analysis in Section 6.3 of our ISCA 2012 paper [14].

5.2. Power and Area

We present the power and area of FR-FCFS and SMS. We find that SMS consumes 66.7% less leakage power than FR-FCFS, which is the simplest of all of the prior memory schedulers that we evaluate. In terms of die area, SMS requires 46.3% less area than FR-FCFS. The majority of the power and area savings of SMS over FR-FCFS come from the decentralized request buffer queues and simpler scheduling logic in SMS. In comparison, FR-FCFS requires centralized request buffer queues, content-addressable memory (CAMs), and complex scheduling logic. Because ATLAS and TCM require more complex ranking and scheduling logic than FR-FCFS, we expect that SMS also provides power and area reductions over ATLAS and TCM.

We provide the following additional results in our ISCA 2012 paper [14]:

- Combined performance of CPU-GPU heterogeneous systems for different SMS configurations with different Shortest Job First (SJF) probability.
- Sensitivity analysis to SMS’s configuration parameters.
- Performance of SMS in CPU-only systems.

6. Related Work

To our knowledge, our ISCA 2012 paper is the first to provide a fundamentally new memory controller design for

heterogeneous CPU-GPU systems in order to reduce interference at the shared off-chip main memory. There are several prior works that reduce interference at the shared off-chip main memory in other systems. We provide a brief discussion of these works.

6.1. Memory Partitioning Techniques

Instead of mitigating the interference problem between applications by scheduling requests at the memory controller, Awasthi et al. [18] propose a mechanism that spreads data in the same working set across memory channels in order to increase memory level parallelism. Memory channel partitioning (MCP) [137] maps applications to different memory channels based on their memory intensities and row buffer locality, to reduce inter-application interference. Mao et al. [128] propose to partition GPU channels and allow only a subset of threads to access each memory channel. In addition to channel partitioning, several works [74, 122, 210] also propose to partition DRAM banks to improve performance. These partitioning techniques are orthogonal to our proposals, and can be combined with SMS to improve the performance of heterogeneous CPU-GPU systems.

6.2. Memory Scheduling Techniques

Memory Scheduling on CPUs. Numerous prior works propose memory scheduling algorithms for CPUs that improve system performance. The first-come-first-serve (FR-FCFS) scheduler [173, 220] prioritizes requests that hit in the row buffer over requests that miss in the row buffer, with the aim of reducing the number of times rows must be activated (as row activation incurs a high latency). Several memory schedulers improve performance beyond FR-FCFS by identifying critical threads in multithreaded applications [47], using reinforcement learning to identify long-term memory behavior [79, 136], prioritizing memory requests based on the criticality (i.e., latency sensitivity) of each memory request [57, 123, 211], distinguishing prefetch requests from demand requests [109, 111], or improving the scheduling of memory writeback requests [110, 182, 193]. While all of these schedulers increase DRAM performance and/or throughput, many of them introduce fairness problems by under-servicing applications that only infrequently issue memory requests. To remedy fairness problems, several application-aware memory scheduling algorithms [98, 99, 135, 141, 142, 194, 195, 197] use information on the memory intensity of each application to balance both performance and fairness. Unlike SMS, none of these schedulers consider the different needs of CPU memory requests and GPU memory requests in a heterogeneous system.

Memory Scheduling on GPUs. Since GPU applications are bandwidth intensive, often with streaming access patterns, a policy that maximizes the number of row buffer hits is effective for GPUs to maximize overall throughput. As a result, FR-FCFS with a large request buffer tends to perform

well for GPUs [22]. In view of this, prior work [213] proposes mechanisms to reduce the complexity of FR-FCFS scheduling for GPUs. Ausavarungnirun et al. [15] propose MeDiC, which is a cache and memory management scheme to improve the performance of GPGPU applications. Jeong et al. [80] propose a QoS-aware memory scheduler that guarantees the performance of GPU applications by prioritizing memory requests from graphics applications over those from CPU applications until the system can guarantee that a frame can be rendered within a given deadline, after which it prioritizes requests from CPU applications. Jog et al. [83] propose CLAM, a memory scheduler that identifies critical memory requests and prioritizes them in the main memory. Ausavarungnirun et al. [17] propose a scheduling algorithm that identifies and prioritizes TLB-related memory requests in GPU-based systems, to reduce the overhead of memory virtualization. Unlike SMS, none of these works holistically optimize the performance *and* fairness of requests when a memory controller is shared by a CPU and a GPU.

Memory Scheduling on Emerging Systems. Recent proposals investigate memory scheduling on emerging platforms. Usui et al. [201, 202] propose accelerator-aware memory controller designs that improve the performance of systems that contain both CPUs and hardware accelerators. Zhao et al. [216] decouple the design of a memory controller for persistent memory into multiple stages. These works build upon principles for heterogeneous system memory scheduling that were first proposed in SMS.

6.3. Other Related Works

DRAM Designs. Aside from memory scheduling and memory partitioning techniques, previous works propose new DRAM designs that are capable of reducing memory latency in conventional DRAM [9, 10, 31, 32, 33, 34, 36, 63, 69, 72, 90, 100, 112, 113, 114, 115, 116, 126, 132, 155, 166, 177, 187, 190, 208, 218] and non-volatile memory [102, 105, 106, 107, 130, 131, 170, 171, 212]. Previous works on bulk data transfer [30, 34, 59, 60, 75, 81, 86, 124, 180, 183, 215, 217] and in-memory computation [7, 8, 11, 19, 25, 26, 44, 52, 54, 55, 56, 58, 61, 70, 71, 87, 94, 101, 127, 157, 160, 161, 168, 181, 184, 185, 192, 198, 214] can be used improve DRAM bandwidth. Techniques to reduce the overhead of DRAM refresh [5, 6, 20, 24, 95, 118, 121, 146, 156, 169, 204] can be applied to improve the performance of GPU-based systems. Data compression techniques [162, 163, 164, 165, 205] can also be used on the main memory to increase the effective available DRAM bandwidth. All of these techniques can mitigate the performance impact of memory interference and improve the performance of GPU-based systems. They are orthogonal to, and can be combined with, SMS to further improve the performance of heterogeneous CPU-GPU systems.

Previous works on data prefetching [12, 21, 29, 39, 41, 46, 48, 49, 64, 65, 73, 84, 85, 104, 108, 109, 111, 138, 140, 144, 148, 186, 191] can also be used to mitigate high DRAM latency. However,

these techniques generally increase DRAM bandwidth utilization, which can lead to lower GPU performance.

Other Ways to Improve Performance on Systems with GPUs. Other works have proposed various methods of decreasing memory divergence. These methods range from thread throttling [88, 89, 103, 174] to warp scheduling [117, 129, 147, 174, 175, 219]. While these methods share our goal of reducing memory divergence, none of them exploit *inter-warp* heterogeneity and, as a result, are orthogonal or complementary to our proposal. Our work also makes new observations about memory divergence that are not covered by these works.

7. Significance and Long-Term Impact

SMS exposes the need to redesign components of the memory subsystem to better serve integrated CPU-GPU systems. Systems-on-chip (SoCs) that integrate CPUs and GPUs on the same die are growing rapidly in popularity (e.g., [37, 133, 152, 153]), due to their high energy efficiency and lower costs compared to discrete CPUs and GPUs. As a result, SoCs are commonly used in mobile devices such as smartphones, tablets, and laptops, and are being used in many servers and data centers. We expect that as more powerful CPUs and GPUs are integrated in SoCs, and as the workloads running on the CPUs/GPUs become more memory-intensive, SMS will become even more essential to alleviate the shared memory subsystem bottleneck.

The observations and mechanisms in our ISCA 2012 paper [14] expose several future research problems. We briefly discuss two future research areas below.

Interference Management in Emerging Heterogeneous Systems. Our ISCA 2012 paper [14] considers heterogeneous systems where a CPU executes various general-purpose applications while the GPU executes graphics workloads. Modern heterogeneous systems contain an increasingly diverse set of workloads. For example, programmers can use the GPU in an integrated CPU-GPU system to execute general-purpose applications (known as GPGPU applications). GPGPU applications can have significantly different access patterns from graphics applications, requiring different memory scheduling policies (e.g., [15, 17, 83]). Future work can adapt the mechanisms of SMS to optimize the performance of GPGPU applications.

Many heterogeneous systems are being deployed in mobile or embedded environments, and must ensure that memory requests from some or all of the components of the heterogeneous system meet *real-time deadlines* [91, 92, 201, 202]. Traditionally, applications with real-time deadlines are executed using embedded cores or fixed-function accelerators, which are often integrated into modern SoCs. We believe that the observations and mechanisms in our ISCA 2012 paper [14] can be used and extended to ensure that these deadlines are met. Recent works [201, 202] have shown that the principles

of SMS can be extended to provide deadline-aware memory scheduling for accelerators within heterogeneous systems.

Even though the mechanisms proposed in our ISCA 2012 paper [14] aim to minimize the slowdown caused by interference, they do not provide actual performance guarantees. However, we believe it is possible to combine principles from SMS with prediction mechanisms for memory access latency (e.g., [91, 92, 196, 197]) to provide *hard* performance guarantees for real-time applications, while still ensuring fairness for all applications executing on the heterogeneous system.

Memory Scheduling for Concurrent GPGPU Applications. While SMS allows CPU applications and graphics applications to share DRAM more efficiently, we assume that there is only a single GPU application running at any given point in time. Recent works [16, 82] propose methods to efficiently share the same GPU across multiple concurrently-executing GPGPU applications. We believe that the techniques and observations provided in our ISCA 2012 paper [14] can be applied to reduce the memory interference induced by additional GPGPU applications. Furthermore, as concurrent GPGPU application execution becomes more widespread, the concepts of SMS can be extended to provide prioritization and fairness across multiple GPGPU applications.

Our analysis of memory interference in heterogeneous systems, and our new Staged Memory Scheduler, have inspired a number of subsequent works. These works include significant research on predictable and/or deadline-aware memory scheduling [91, 92, 194, 195, 197, 201, 202, 216], and on other memory scheduling algorithms for heterogeneous systems [161, 201, 202, 207].

8. Conclusion

While many advancements in memory scheduling policies have been made to deal with multi-core processors, the integration of GPUs on the same chip as the CPUs has created new system design challenges. Our ISCA 2012 paper [14] demonstrates how the inclusion of GPU memory traffic can cause severe difficulties for existing memory controller designs in terms of performance and especially fairness. We propose a new approach, Staged Memory Scheduler, which delivers superior performance and fairness for integrated CPU-GPU systems compared to state-of-the-art memory schedulers, while providing a design that is significantly simpler to implement (thus improving the scalability of the memory controller). The key insight behind simplifying the implementation of SMS is that the primary functions of sophisticated memory controller algorithms can be decoupled. As a result, SMS proposes a multi-stage memory controller architecture. We show that SMS significantly improves the performance and fairness in integrated CPU-GPU systems. We hope and expect that our observations and mechanisms can inspire future work in memory system design for existing and emerging heterogeneous systems.

Acknowledgments

We thank Saugata Ghose for his dedicated effort in the preparation of this article. We thank Stephen Somogyi and Fritz Kruger at AMD for their assistance with the modeling of the GPU applications. We also thank Antonio Gonzalez, anonymous reviewers and members of the SAFARI group at CMU for their feedback. We acknowledge the generous support of AMD, Intel, Oracle, and Samsung. This research was also partially supported by grants from the NSF (CAREER Award CCF-0953246 and CCF-1147397), GSRC, and Intel ARO Memory Hierarchy Program.

References

- [1] Advanced Micro Devices, “AMD Accelerated Processing Units.”
- [2] Advanced Micro Devices, “ATI Radeon GPGPUs.”
- [3] Advanced Micro Devices, “AMD Radeon R9 290X,” 2013.
- [4] A. Agarwal, B. H. Lim, D. Kranz, and J. Kubiatowicz, “APRIL: A Processor Architecture for Multiprocessing,” in *ISCA*, 1990.
- [5] A. Agrawal, A. Ansari, and J. Torrellas, “Mosaic: Exploiting the Spatial Locality of Process Variation to Reduce Refresh Energy in On-chip eDRAM Modules,” in *HPCA*, 2014.
- [6] A. Agrawal, M. O’Connor, E. Bolotin, N. Chatterjee, J. Emer, and S. Keckler, “CLARA: Circular Linked-List Auto and Self Refresh Architecture,” in *MEMSYS*, 2016.
- [7] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, “A Scalable Processing-in-memory Accelerator for Parallel Graph Processing,” in *ISCA*, 2015.
- [8] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, “PIM-enabled Instructions: A Low-overhead, Locality-aware Processing-in-memory Architecture,” in *ISCA*, 2015.
- [9] J. H. Ahn, J. Leverich, R. Schreiber, and N. P. Jouppi, “Multicore DIMM: an Energy Efficient Memory Module with Independently Controlled DRAMs,” *IEEE CAL*, 2009.
- [10] J. H. Ahn, N. P. Jouppi, C. Kozyrakis, J. Leverich, and R. S. Schreiber, “Improving System Energy Efficiency with Memory Rank Subsetting,” *ACM TACO*, vol. 9, no. 1, pp. 4:1–4:28, 2012.
- [11] B. Akin, F. Franchetti, and J. C. Hoe, “Data Reorganization in Memory Using 3D-stacked DRAM,” in *ISCA*, 2015.
- [12] A. R. Alameldeen and D. A. Wood, “Interactions Between Compression and Prefetching in Chip Multiprocessors,” in *HPCA*, 2007.
- [13] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, “The Tera Computer System,” in *ICS*, 1990.
- [14] R. Ausavarungnirun, K. Chang, L. Subramanian, G. Loh, and O. Mutlu, “Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems,” in *ISCA*, 2012.
- [15] R. Ausavarungnirun, S. Ghose, O. Kayiran, G. H. Loh, C. R. Das, M. T. Kandemir, and O. Mutlu, “Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance,” in *PACT*, 2015.
- [16] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu, “Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes,” in *MICRO*, 2017.
- [17] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. J. Rossbach, and O. Mutlu, “MASK: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency,” in *ASPLOS*, 2018.
- [18] M. Awasthi, D. W. Nellans, K. Sudan, R. Balasubramanian, and A. Davis, “Handling the Problems and Opportunities Posed by Multiple On-chip Memory Controllers,” in *PACT*, 2010.
- [19] O. O. Babarinsa and S. Idreos, “JAFAR: Near-Data Processing for Databases,” in *SIGMOD*, 2015.
- [20] S. Baek, S. Cho, and R. Melhem, “Refresh Now and Then,” *IEEE TC*, vol. 63, no. 12, pp. 3114–3126, 2014.
- [21] J.-L. Baer and T.-F. Chen, “Effective Hardware-Based Data Prefetching for High-Performance Processors,” *IEEE TC*, vol. 44, no. 5, pp. 609–623, 1995.
- [22] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, “Analyzing CUDA Workloads Using a Detailed GPU Simulator,” in *ISPASS*, 2009.
- [23] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes, “The Illiac IV Computer,” *IEEE TC*, vol. 100, no. 8, pp. 746–757, 1968.
- [24] I. Bhati, Z. Chishti, S.-L. Lu, and B. Jacob, “Flexible Auto-refresh: Enabling Scalable and Energy-efficient DRAM Refresh Reductions,” in *ISCA*, 2015.
- [25] A. Boroumand *et al.*, “Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks,” in *ASPLOS*, 2018.
- [26] A. Boroumand, S. Ghose, B. Lucia, K. Hsieh, K. Malladi, H. Zheng, and O. Mutlu, “LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory,” *IEEE CAL*, 2016.
- [27] D. Bouvier and B. Sander, “Applying AMD’s ‘Kaveri’ APU for Heterogeneous Computing,” in *Hot Chips*, 2014.
- [28] B. Burgess, B. Cohen, J. Dundas, J. Rupley, D. Kaplan, and M. Denman, “Bobcat: AMD’s Low-Power x86 Processor,” *IEEE Micro*, 2011.
- [29] P. Cao, E. W. Felten, A. R. Karlin, and K. Li, “A Study of Integrated Prefetching and Caching Strategies,” in *SIGMETRICS*, 1995.
- [30] J. Carter *et al.*, “Impulse: Building a Smarter Memory Controller,” in *HPCA*, 1999.
- [31] K. Chandrasekar, S. Goossens, C. Weis, M. Koedam, B. Akesson, N. Wehn, and K. Goossens, “Exploiting Expendable Process-Margins in DRAMs for Run-Time Performance Optimization,” in *DATE*, 2014.
- [32] K. Chang, A. Kashyap, H. Hassan, S. Ghose, K. Hsieh, D. Lee, T. Li, G. Pekhimenko, S. Khan, and O. Mutlu, “Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization,” in *SIGMETRICS*, 2016.
- [33] K. Chang, D. Lee, Z. Chishti, A. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu, “Improving DRAM Performance by Parallelizing Refreshes with Accesses,” in *HPCA*, 2014.
- [34] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu, “Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM,” in *HPCA*, 2016.
- [35] K. K. Chang, A. G. Yaglikci, S. Ghose, A. Agrawal, N. Chatterjee, A. Kashyap, D. Lee, M. O’Connor, H. Hassan, and O. Mutlu, “Understanding Reduced-Voltage Operation in Modern DRAM Devices: Experimental Characterization, Analysis, and Mechanisms,” in *SIGMETRICS*, 2017.
- [36] N. Chatterjee, M. Shevgoor, R. Balasubramanian, A. Davis, Z. Fang, R. Illikkal, and R. Iyer, “Leveraging Heterogeneity in DRAM Main Memories to Accelerate Critical Word Access,” in *MICRO*, 2012.
- [37] M. Clark, “A New X86 Core Architecture for the Next Generation of Computing,” in *Hot Chips*, 2016.
- [38] Control Data Corporation, “Control Data 7600 Computer Systems Reference Manual,” 1972.
- [39] R. Cooksey, S. Jourdan, and D. Grunwald, “A Stateless, Content-directed Data Prefetching Mechanism,” in *ASPLOS*, 2002.
- [40] B. A. Crane and J. A. Githens, “Bulk Processing in Distributed Logic Memory,” *IEEE EC*, 1965.
- [41] F. Dahlgren, M. Dubois, and P. Stenström, “Sequential Hardware Prefetching in Shared-Memory Multiprocessors,” *IEEE TPDS*, vol. 6, no. 7, pp. 733–746, 1995.
- [42] R. Das, R. Ausavarungnirun, O. Mutlu, A. Kumar, and M. Azimi, “Application-to-core Mapping Policies to Reduce Memory System Interference in Multi-core Systems,” in *HPCA*, 2013.
- [43] R. Das, O. Mutlu, T. Moscibroda, and C. R. Das, “Application-Aware Prioritization Mechanisms for On-Chip Networks,” in *MICRO*, 2009.
- [44] J. Draper *et al.*, “The Architecture of the DIVA Processing-in-memory Chip,” in *ICS*, 2002.
- [45] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, “Fairness via Source Throttling: A Configurable and High-performance Fairness Substrate for Multi-core Memory Systems,” in *ASPLOS*, 2010.
- [46] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, “Prefetch-aware Shared Resource Management for Multi-core Systems,” in *ISCA*, 2011.
- [47] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt, “Parallel Application Memory Scheduling,” in *MICRO*, 2011.
- [48] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, “Coordinated Control of Multiple Prefetchers in Multi-core Systems,” in *MICRO*, 2009.
- [49] E. Ebrahimi, O. Mutlu, and Y. N. Patt, “Techniques for Bandwidth-efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems,” in *HPCA*, 2009.
- [50] S. Eyerman and L. Eeckhout, “System-Level Performance Metrics for Multiprogram Workloads,” *IEEE Micro*, 2008.
- [51] S. Eyerman and L. Eeckhout, “Restating the Case for Weighted-IPC Metrics to Evaluate Multiprogram Workload Performance,” *IEEE CAL*, 2014.
- [52] A. Farnahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, “NDA: Near-DRAM Acceleration Architecture Leveraging Commodity DRAM Devices and Standard Memory Modules,” in *HPCA*, 2015.
- [53] M. Flynn, “Very High-Speed Computing Systems,” *Proc. of the IEEE*, vol. 54, no. 2, 1966.
- [54] B. B. Fraguola, J. Renau, P. Feautrier, D. Padua, and J. Torrellas, “Programming the FlexRAM Parallel Intelligent Memory System,” in *PPoPP*, 2003.
- [55] M. Gao, G. Ayers, and C. Kozyrakis, “Practical Near-Data Processing for In-Memory Analytics Frameworks,” in *PACT*, 2015.
- [56] M. Gao and C. Kozyrakis, “HRL: Efficient and Flexible Reconfigurable Logic for Near-data Processing,” in *HPCA*, 2016.
- [57] S. Ghose, H. Lee, and J. F. Martinez, “Improving Memory Scheduling via Processor-side Load Criticality Information,” in *ISCA*, 2013.
- [58] M. Gokhale, B. Holmes, and K. Iobst, “Processing in Memory: the Terasys Massively Parallel PIM Array,” *Computer*, vol. 28, no. 4, pp. 23–31, 1995.
- [59] M. Gschwind, “Chip Multiprocessing and the Cell Broadband Engine,” in *CF*, 2006.
- [60] J. Gummaraju, M. Erez, J. Coburn, M. Rosenblum, and W. J. Dally, “Architectural Support for the Stream Execution Model on General-Purpose Processors,” in *PACT*, 2007.
- [61] Q. Guo, N. Alachiotis, B. Akin, F. Sadi, G. Xu, T.-M. Low, L. Pileggi, J. C. Hoe, and F. Franchetti, “3D-Stacked Memory-Side Acceleration: Accelerator and System Design,” in *WONDP*, 2014.

- [62] R. H. Halstead and T. Fujita, "MASA: A Multithreaded Processor Architecture for Parallel Symbolic Computing," in *ISCA*, 1988.
- [63] C. A. Hart, "CDRAM in a Unified Memory Architecture," in *Intl. Computer Conference*, 1994.
- [64] M. Hashemi, O. Mutlu, and Y. N. Patt, "Continuous Runahead: Transparent Hardware Acceleration for Memory Intensive Workloads," in *MICRO*, 2016.
- [65] M. Hashemi, Khubaib, E. Ebrahimi, O. Mutlu, and Y. N. Patt, "Accelerating Dependent Cache Misses with an Enhanced Memory Controller," in *ISCA*, 2016.
- [66] H. Hassan, G. Pekhimenko, N. Vijaykumar, V. Seshadri, D. Lee, O. Ergin, and O. Mutlu, "ChargeCache: Reducing DRAM Latency by Exploiting Row Access Locality," in *HPCA*, 2016.
- [67] H. Hassan, N. Vijaykumar, S. Khan, S. Ghose, K. Chang, G. Pekhimenko, D. Lee, O. Ergin, and O. Mutlu, "SoftMC: A Flexible and Practical Open-source Infrastructure for Enabling Experimental DRAM Studies," in *HPCA*, 2017.
- [68] H. Hellerman, "Parallel Processing of Algebraic Expressions," *IEEE Transactions on Electronic Computers*, 1966.
- [69] H. Hidaka, Y. Matsuda, M. Asakura, and K. Fujishima, "The Cache DRAM Architecture," *IEEE Micro*, 1990.
- [70] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu, "Accelerating Pointer Chasing in 3D-stacked Memory: Challenges, Mechanisms, Evaluation," in *ICCD*, 2016.
- [71] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler, "Transparent Offloading and Mapping (TOM): Enabling Programmer-transparent Near-data Processing in GPU Systems," in *ISCA*, 2016.
- [72] W.-C. Hsu and J. E. Smith, "Performance of Cached DRAM Organizations in Vector Supercomputers," in *ISCA*, 1993.
- [73] I. Hur and C. Lin, "Memory Prefetching Using Adaptive Stream Detection," in *MICRO*, 2006.
- [74] T. Ikeda and K. Kise, "Application Aware DRAM Bank Partitioning in CMP," in *ICPADS*, 2013.
- [75] Intel Corp., "Intel® I/O Acceleration Technology."
- [76] Intel Corp., "Sandy Bridge Intel Processor Graphics Performance Developer's Guide," <https://software.intel.com/en-us/articles/intel-snbgraphics-developers-guides>, 2012.
- [77] Intel Corp., "Introduction to Intel® Architecture," 2014.
- [78] Intel Corp., "6th Generation Intel® Core™ Processor Family Datasheet, Vol. 1," 2017.
- [79] E. Ipek, O. Mutlu, J. F. Martinez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," in *ISCA*, 2008.
- [80] M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver, "A QoS-Aware Memory Controller for Dynamically Balancing GPU and CPU Bandwidth Use in an MPSoC," in *DAC*, 2012.
- [81] X. Jiang, Y. Solihin, L. Zhao, and R. Iyer, "Architecture Support for Improving Bulk Memory Copying and Initialization Performance," in *PACT*, 2009.
- [82] A. Jog, O. Kayiran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S. W. Keckler, M. T. Kandemir, and C. R. Das, "Anatomy of GPU Memory System for Multi-Application Execution," in *MEMSYS*, 2015.
- [83] A. Jog, O. Kayiran, A. Pattnaik, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Exploiting Core Criticality for Enhanced GPU Performance," in *SIGMETRICS*, 2016.
- [84] D. Joseph and D. Grunwald, "Prefetching Using Markov Predictors," in *ISCA*, 1997.
- [85] N. P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," in *ISCA*, 1990.
- [86] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the Cell Multiprocessor," *IBM JRD*, 2005.
- [87] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas, "FlexRAM: Toward an Advanced Intelligent Memory System," in *ICCD*, 1999.
- [88] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither More Nor Less: Optimizing Thread-Level Parallelism for GPGPUs," in *PACT*, 2013.
- [89] O. Kayiran, N. C. Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das, "Managing GPU Concurrency in Heterogeneous Architectures," in *MICRO*, 2014.
- [90] G. Kedem and R. P. Koganti, "WCDRAM: A Fully Associative Integrated Cached-DRAM with Wide Cache Lines," Duke Univ. Dept. of Computer Science, Tech. Rep. CS-1997-03, 1997.
- [91] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, "Bounding Memory Interference Delay in COTS-Based Multi-Core Systems," in *RTAS*, 2014.
- [92] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, "Bounding and Reducing Memory Interference in COTS-Based Multi-Core Systems," *RTS*, 2016.
- [93] J. S. Kim, M. Patel, H. Hassan, and O. Mutlu, "The DRAM Latency PUF: Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency-Reliability Tradeoff in Modern DRAM Devices," in *HPCA*, 2018.
- [94] J. S. Kim, D. Senol, H. Xin, D. Lee, S. Ghose, M. Alser, H. Hassan, O. Ergin, C. Alkan, and O. Mutlu, "GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-in-Memory Technologies," *BMC Genomics*, 2018.
- [95] J. Kim and M. C. Papaefthymiou, "Block-based Multi-period Refresh for Energy Efficient Dynamic Memory," in *ASIC*, 2001.
- [96] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," *CAL*, 2015.
- [97] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," in *ISCA*, 2014.
- [98] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers," in *HPCA*, 2010.
- [99] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *MICRO*, 2010.
- [100] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM," in *ISCA*, 2012.
- [101] P. M. Kogge, "EXECUBE-A New Architecture for Scalable MPPs," in *ICPP*, 1994.
- [102] E. Kultursay, M. Kandemir, A. Sivasubramanian, and O. Mutlu, "Evaluating STT-RAM as an energy-efficient main memory alternative," in *ISPASS*, 2013.
- [103] H.-K. Kuo, B. C. Lai, and J.-Y. Jou, "Reducing Contention in Shared Last-Level Cache for Throughput Processors," *ACM TODAES*, vol. 20, no. 1, 2014.
- [104] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-block Prediction & Dead-block Correlating Prefetchers," in *ISCA*, 2001.
- [105] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting Phase Change Memory as a Scalable DRAM Alternative," in *ISCA*, 2009.
- [106] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Phase Change Memory Architecture and the Quest for Scalability," *CACM*, vol. 53, no. 7, pp. 99–106, 2010.
- [107] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-Change Technology and the Future of Main Memory," *IEEE Micro*, vol. 30, no. 1, pp. 143–143, 2010.
- [108] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt, "Prefetch-aware Memory Controllers," *IEEE TC*, vol. 60, no. 10, pp. 1406–1430, 2011.
- [109] C. J. Lee, V. Narasiman, O. Mutlu, and Y. N. Patt, "Improving Memory Bank-Level Parallelism in the Presence of Prefetching," in *MICRO*, 2009.
- [110] C. J. Lee, E. Ebrahimi, V. Narasiman, O. Mutlu, and Y. N. Patt, "DRAM-Aware Last-Level Cache Writeback: Reducing Write-Caused Interference in Memory Systems," Univ. of Texas at Austin, High Performance Systems Group, Tech. Rep. TR-HPSS-2010-002, 2010.
- [111] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt, "Prefetch-aware DRAM Controllers," in *MICRO*, 2008.
- [112] D. Lee, L. Subramanian, R. Ausavarungnirun, J. Choi, and O. Mutlu, "Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM," in *PACT*, 2015.
- [113] D. Lee, S. Ghose, G. Pekhimenko, S. Khan, and O. Mutlu, "Simultaneous Multi-layer Access: Improving 3D-stacked Memory Bandwidth at Low Cost," *TACO*, 2016.
- [114] D. Lee, S. Khan, L. Subramanian, S. Ghose, R. Ausavarungnirun, G. Pekhimenko, V. Seshadri, and O. Mutlu, "Design-Induced Latency Variation in Modern DRAM Chips: Characterization, Analysis, and Latency Reduction Mechanisms," in *SIGMETRICS*, 2017.
- [115] D. Lee, Y. Kim, G. Pekhimenko, S. Khan, V. Seshadri, K. Chang, and O. Mutlu, "Adaptive-latency DRAM: Optimizing DRAM Timing for the Common-case," in *HPCA*, 2015.
- [116] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu, "Tiered-latency DRAM: A Low Latency and Low Cost DRAM Architecture," in *HPCA*, 2013.
- [117] S.-Y. Lee and C.-J. Wu, "CAWS: Criticality-Aware Warp Scheduling for GPGPU Workloads," in *PACT*, 2014.
- [118] C. H. Lin, D. Y. Shen, Y. J. Chen, C. L. Yang, and M. Wang, "SECRET: Selective Error Correction for Refresh Energy Reduction in DRAMs," in *ICCD*, 2012.
- [119] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro*, 2008.
- [120] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu, "An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms," in *ISCA*, 2013.
- [121] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, "RAIDR: Retention-aware Intelligent DRAM Refresh," in *ISCA*, 2012.
- [122] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu, "A Software Memory Partitioning Approach for Eliminating Bank-level Interference in Multicore Systems," in *PACT*, 2012.
- [123] W. Liu, P. Huang, T. Kun, T. Lu, K. Zhou, C. Li, and X. He, "LAMs: A Latency-aware Memory Scheduling Policy for Modern DRAM Systems," in *IPCCC*, 2016.
- [124] S.-L. Lu, Y.-C. Lin, and C.-L. Yang, "Improving DRAM Latency with Dynamic Asymmetric Subarray," in *MICRO*, 2015.
- [125] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *PLDI*, 2005.
- [126] Y. Luo, S. Govindan, B. Sharma, M. Santaniello, J. Meza, A. Kansal, J. Liu, B. Khesib, K. Vaid, and O. Mutlu, "Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-Reliability Memory," in *DSN*, 2014.
- [127] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, "Smart Memories: A Modular Reconfigurable Architecture," in *ISCA*, 2000.
- [128] M. Mao, W. Wen, X. Liu, J. Hu, D. Wang, Y. Chen, and H. Li, "TEMP: Thread Batch Enabled Memory Partitioning for GPU," in *DAC*, 2016.

- [129] J. Meng, D. Tarjan, and K. Skadron, "Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance," in *ISCA*, 2010.
- [130] J. Meza, Y. Luo, S. Khan, J. Zhao, Y. Xie, and O. Mutlu, "A Case for Efficient Hardware/Software Cooperative Management of Storage and Memory," in *WEED*, 2013.
- [131] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan, "Enabling Efficient and Scalable Hybrid Memories Using Fine-Granularity DRAM Cache Management," *IEEE CAL*, 2012.
- [132] Micron Technology, Inc., "576Mb: x18, x36 RDRAM3," 2011.
- [133] R. Mijat, "Take GPU Processing Power Beyond Graphics with Mali GPU Computing," 2012.
- [134] T. Moscibroda and O. Mutlu, "Memory Performance Attacks: Denial of Memory Service in Multi-core Systems," in *USENIX Security*, 2007.
- [135] T. Moscibroda and O. Mutlu, "Distributed Order Scheduling and Its Application to Multi-core DRAM Controllers," in *PODC*, 2008.
- [136] J. Mukundan and J. F. Martinez, "MORSE: Multi-objective Reconfigurable Self-optimizing Memory Scheduler," in *HPCA*, 2012.
- [137] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, "Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning," in *MICRO*, 2011.
- [138] O. Mutlu, H. Kim, and Y. N. Patt, "Address-value Delta (AVD) Prediction: Increasing the Effectiveness of Runahead Execution by Exploiting Regular Memory Allocation Patterns," in *MICRO*, 2005.
- [139] O. Mutlu, "Memory scaling: A systems architecture perspective," in *IMW*, 2013.
- [140] O. Mutlu, H. Kim, and Y. N. Patt, "Techniques for Efficient Processing in Runahead Execution Engines," in *ISCA*, 2005.
- [141] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *MICRO*, 2007.
- [142] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems," in *ISCA*, 2008.
- [143] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enabling High-Performance and Fair Memory Controllers," *IEEE Micro*, 2009.
- [144] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors," in *HPCA*, 2003.
- [145] O. Mutlu and L. Subramanian, "Research Problems and Opportunities in Memory Systems," *SUPERFRI*, 2014.
- [146] P. J. Nair, D.-H. Kim, and M. K. Qureshi, "ArchShield: Architectural Framework for Assisting DRAM Scaling by Tolerating High Error Rates," in *ISCA*, 2013.
- [147] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling," in *MICRO*, 2011.
- [148] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith, "AC/DC: An Adaptive Data Cache Prefetcher," in *PACT*, 2004.
- [149] NVIDIA Corp., "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf, 2011.
- [150] NVIDIA Corp., "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," 2012.
- [151] NVIDIA Corp., "NVIDIA GeForce GTX 750 Ti," 2014.
- [152] NVIDIA Corp., "NVIDIA Tegra K1," http://www.nvidia.com/content/pdf/tegra_white_papers/tegra-k1-whitepaper-v1.0.pdf, 2014.
- [153] NVIDIA Corp., "NVIDIA Tegra X1," <https://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf>, 2015.
- [154] NVIDIA Corp., "NVIDIA Tesla P100," 2016.
- [155] S. O. Y. H. Son, N. S. Kim, and J. H. Ahn, "Row-Buffer Decoupling: A Case for Low-Latency DRAM Microarchitecture," in *ISCA*, 2014.
- [156] T. Ohsawa, K. Kai, and K. Murakami, "Optimizing the DRAM Refresh Count for Merged DRAM/Logic LSI's," in *ISLPED*, 1998.
- [157] M. Oskin, F. T. Chong, and T. Sherwood, "Active Pages: A Computation Model for Intelligent Memory," in *ISCA*, 1998.
- [158] M. Patel, J. S. Kim, and O. Mutlu, "The Reach Profiler (REAPER): Enabling the Mitigation of DRAM Retention Failures via Profiling at Aggressive Conditions," in *ISCA*, 2017.
- [159] 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*, 2004.
- [160] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A Case for Intelligent RAM," *IEEE Micro*, vol. 17, no. 2, pp. 34–44, 1997.
- [161] A. Pattanaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das, "Scheduling Techniques for GPU Architectures with Processing-In-Memory Capabilities," in *PACT*, 2016.
- [162] G. Pekhimenko, E. Bolotin, N. Vijaykumar, O. Mutlu, T. C. Mowry, and S. W. Keckler, "A Case for Toggle-aware Compression for GPU Systems," in *HPCA*, 2016.
- [163] G. Pekhimenko, T. Huberty, R. Cai, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Exploiting Compressed Block Size as an Indicator of Future Reuse," in *HPCA*, 2015.
- [164] G. Pekhimenko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry, "Linearly Compressed Pages: A Main Memory Compression Framework with Low Complexity and Low Latency," in *MICRO*, 2013.
- [165] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate Compression: Practical Data Compression for On-chip Caches," in *PACT*, 2012.
- [166] S. Phadke and S. Narayanasamy, "MLP Aware Heterogeneous Memory System," in *DATE*, 2011.
- [167] PowerVR, "PowerVR Hardware Architecture Overview for Developers," <http://cdn.imgtec.com/sdk-documentation/PowerVR+Hardware+Architecture+Overview+for+Developers.pdf>, 2016.
- [168] S. H. Pugsley, J. Jests, H. Zhang, R. Balasubramanian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "NDC: Analyzing the Impact of 3D-stacked Memory+logic Devices on MapReduce Workloads," in *ISPASS*, 2014.
- [169] M. K. Qureshi, D. H. Kim, S. Khan, P. J. Nair, and O. Mutlu, "AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems," in *DSN*, 2015.
- [170] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing Lifetime and Security of PCM-based Main Memory with Start-gap Wear Leveling," in *MICRO*, 2009.
- [171] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable High Performance Main Memory System Using Phase-change Memory Technology," in *ISCA*, 2009.
- [172] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn, "PIN: A Binary Instrumentation Tool for Computer Architecture Research and Education," in *WCAE*, 2004.
- [173] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory Access Scheduling," in *ISCA*, 2000.
- [174] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *MICRO*, 2012.
- [175] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Divergence-Aware Warp Scheduling," in *MICRO*, 2013.
- [176] R. M. Russell, "The CRAY-1 Computer System," *CACM*, vol. 21, no. 1, pp. 63–72, 1978.
- [177] Y. Sato *et al.*, "Fast cycle RAM (FCRAM): A 20-ns Random Row Access, Pipeline-Operating DRAM," in *VLSIC*, 1998.
- [178] P. B. Schneek, *The CDC STAR-100*. Boston, MA: Springer US, 1987, pp. 99–117. http://dx.doi.org/10.1007/978-1-4615-7957-1_5
- [179] D. N. Senzig and R. V. Smith, "Computer Organization for Array Processing," in *AIPS*, 1965.
- [180] S.-Y. Seo, "Methods of Copying a Page in a Memory Device and Methods of Managing Pages in a Memory System," U.S. Patent Application 20140185395, 2014.
- [181] V. Seshadri, K. Hsieh, A. Boroumand, D. Lee, M. Kozuch, O. Mutlu, P. Gibbons, and T. C. Mowry, "Fast Bulk Bitwise AND and OR in DRAM," *IEEE CAL*, 2015.
- [182] V. Seshadri, A. Bhowmick, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "The Dirty-Block Index," in *ISCA*, 2014.
- [183] V. Seshadri *et al.*, "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization," in *ISCA*, 2013.
- [184] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-memory Accelerator for Bulk Bitwise Operations using Commodity DRAM Technology," in *MICRO*, 2017.
- [185] V. Seshadri, T. Mullins, A. Boroumand, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Gather-Scatter DRAM: In-DRAM Address Translation to Improve the Spatial Locality of Non-Unit Strided Accesses," in *MICRO*, 2015.
- [186] V. Seshadri, S. Yedkar, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Mitigating Prefetcher-Caused Pollution Using Informed Caching Policies for Prefetched Blocks," *ACM TACO*, vol. 11, no. 4, pp. 51:1–51:22, 2015.
- [187] W. Shin, J. Yang, J. Choi, and L.-S. Kim, "NUAT: A Non-Uniform Access Time Memory Controller," in *HPCA*, 2014.
- [188] D. L. Slotnick, W. C. Borck, and R. C. McReynolds, "The Solomon Computer – A Preliminary Report," in *Workshop on Computer Organization*, 1962.
- [189] B. Smith, "Architecture and Applications of the HEP Multiprocessor Computer System," *SPIE*, 1981.
- [190] Y. H. Son, S. O. Y. Ro, J. W. Lee, and J. H. Ahn, "Reducing Memory Access Latency with Asymmetric DRAM Bank Organizations," in *ISCA*, 2013.
- [191] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers," in *HPCA*, 2007.
- [192] H. S. Stone, "A Logic-in-Memory Computer," *IEEE TC*, vol. C-19, no. 1, pp. 73–78, 1970.
- [193] 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*, 2010.
- [194] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, "BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling," in *IEEE TPDS*, 2016.
- [195] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, "The Blacklisting Memory Scheduler: Achieving high performance and fairness at low cost," in *ICCD*, 2014.
- [196] L. Subramanian *et al.*, "The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory," in *MICRO*, 2015.
- [197] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu, "MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems," in *HPCA*, 2013.
- [198] Z. Sura *et al.*, "Data Access Optimization in a Processing-in-memory System," in *CF*, 2015.

- [199] J. E. Thornton, "Parallel Operation in the Control Data 6600," in *AFIPS FJCC*, 1964.
- [200] J. E. Thornton, *Design of a Computer—The Control Data 6600*. Scott Foresman & Co, 1970.
- [201] H. Usui, L. Subramanian, K. Chang, and O. Mutlu, "SQUASH: Simple qos-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators," *arXiv CoRR*, 2015.
- [202] H. Usui, L. Subramanian, K. Chang, and O. Mutlu, "DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators," *ACM TACO*, vol. 12, no. 4, Jan. 2016.
- [203] H. Vandierendonck and A. Seznec, "Fairness Metrics for Multi-threaded Processors," *IEEE CAL*, Feb 2011.
- [204] R. Venkatesan, S. Herr, and E. Rotenberg, "Retention-aware Placement in DRAM (RAPID): Software Methods for Quasi-non-volatile DRAM," in *HPCA*, 2006.
- [205] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungnirun, C. Das, M. Kandemir, T. C. Mowry, and O. Mutlu, "A Case for Core-Assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps," in *ISCA*, 2015.
- [206] Vivante, "Vivante Vega GPGPU Technology," <http://www.vivantecorp.com/index.php/en/technology/gpgpu.html>, 2016.
- [207] H. Wang, R. Singh, M. J. Schulte, and N. S. Kim, "Memory Scheduling Towards High-throughput Cooperative Heterogeneous Computing," in *PACT*, 2014.
- [208] F. A. Ware and C. Hampel, "Improving Power and Data Efficiency with Threaded Memory Modules," in *ICCD*, 2006.
- [209] S. Wasson. (2011, Oct.) AMD's A8-3800 Fusion APU.
- [210] M. Xie, D. Tong, K. Huang, and X. Cheng, "Improving System Throughput and Fairness Simultaneously in Shared Memory CMP Systems via Dynamic Bank Partitioning," in *HPCA*, 2014.
- [211] D. Xiong, K. Huang, X. Jiang, and X. Yan, "Memory Access Scheduling Based on Dynamic Multilevel Priority in Shared DRAM Systems," *ACM TACO*, vol. 13, no. 4, Dec. 2016.
- [212] H. Yoon, J. Meza, R. Ausavarungnirun, R. Harding, and O. Mutlu, "Row Buffer Locality Aware Caching Policies for Hybrid Memories," in *ICCD*, 2012.
- [213] G. Yuan, A. Bakhoda, and T. Aamodt, "Complexity Effective Memory Access Scheduling for Many-Core Accelerator Architectures," in *MICRO*, 2009.
- [214] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, "TOP-PIM: Throughput-oriented Programmable Processing in Memory," in *HPDC*, 2014.
- [215] L. Zhang, Z. Fang, M. Parker, B. K. Mathew, L. Schaelicke, J. B. Carter, W. C. Hsieh, and S. A. McKee, "The Impulse Memory Controller," *IEEE TC*, vol. 50, no. 11, pp. 1117–1132, 2001.
- [216] J. Zhao, O. Mutlu, and Y. Xie, "FIRM: Fair and High-Performance Memory Control for Persistent Memory Systems," in *MICRO*, 2014.
- [217] L. Zhao, R. Iyer, S. Makineni, L. Bhuyan, and D. Newell, "Hardware Support for Bulk Data Movement in Server Platforms," in *ICCD*, 2005.
- [218] H. Zheng, J. Lin, Z. Zhang, E. Gorbato, H. David, and Z. Zhu, "Mini-rank: Adaptive DRAM Architecture for Improving Memory Power Efficiency," in *MICRO*, 2008.
- [219] Z. Zheng, Z. Wang, and M. Lipasti, "Adaptive Cache and Concurrency Allocation on GPGPUs," *IEEE CAL*, 2014.
- [220] W. K. Zuravleff and T. Robinson, "Controller for a Synchronous DRAM That Maximizes Throughput by Allowing Memory Requests and Commands to Be Issued Out of Order," US Patent No. 5,630,096, 1997.

Exploiting the DRAM Microarchitecture to Increase Memory-Level Parallelism

Yoongu Kim¹

Vivek Seshadri^{2,1}

Donghyuk Lee^{3,1}

Jamie Liu^{4,1}

Onur Mutlu^{5,1}

¹Carnegie Mellon University

²Microsoft Research India

³NVIDIA Research

⁴Google

⁵ETH Zürich

This paper summarizes the idea of Subarray-Level Parallelism (SALP) in DRAM, which was published in ISCA 2012 [66], and examines the work’s significance and future potential. Modern DRAMs have multiple banks to serve multiple memory requests in parallel. However, when two requests go to the same bank, they have to be served serially, exacerbating the high latency of off-chip memory. Adding more banks to the system to mitigate this problem incurs high system cost. Our goal in this work is to achieve the benefits of increasing the number of banks with a low-cost approach. To this end, we propose three new mechanisms, SALP-1, SALP-2, and MASA (Multitude of Activated Subarrays), to reduce the serialization of different requests that go to the same bank. The key observation exploited by our mechanisms is that a modern DRAM bank is implemented as a collection of subarrays that operate largely independently while sharing few global peripheral structures.

Our three proposed mechanisms mitigate the negative impact of bank serialization by overlapping different components of the bank access latencies of multiple requests that go to different subarrays within the same bank. SALP-1 requires no changes to the existing DRAM structure, and needs to only reinterpret some of the existing DRAM timing parameters. SALP-2 and MASA require only modest changes ($< 0.15\%$ area overhead) to the DRAM peripheral structures, which are much less design constrained than the DRAM core. Our evaluations show that SALP-1, SALP-2 and MASA significantly improve performance for both single-core systems (7%/13%/17%) and multi-core systems (15%/16%/20%), averaged across a wide range of workloads. We also demonstrate that our mechanisms can be combined with application-aware memory request scheduling in multi-core systems to further improve performance and fairness.

Our proposed technique has enabled significant research in the use of subarrays for various purposes (e.g., [15, 16, 21, 37, 76, 78, 84, 87, 128, 129, 130, 135, 156, 159]). SALP has also been described and evaluated by a recent work by Samsung and Intel [54] as a promising mechanism to tolerate long write latencies that are a result of aggressive DRAM technology scaling.

1. Introduction

To be able to serve multiple memory requests in parallel, modern DRAM chips employ multiple banks that can be accessed independently, providing bank level parallelism. Unfortunately, if two memory requests go to the same bank, they have to be served one after another. This is called a *bank conflict*. In the worst case, bank conflicts may delay

a memory request by hundreds or even thousands of nanoseconds [16, 37, 66, 129]. In particular, bank conflicts cause three specific problems that degrade the access latency, bandwidth utilization, and energy efficiency of the main memory subsystem:

1. **Serialization.** Bank conflicts serialize requests that could potentially have been served in parallel. Such serialization exacerbates the already large latency of a memory access, and may cause processor cores to stall for much longer.
2. **Write Recovery.** A request scheduled after a write request to the same bank experiences an extra delay called the *write recovery penalty*, which is an additional time required to safely store new data in the cells. This write recovery latency further aggravates the impact of serialization.
3. **Row Buffer Thrashing.** Each bank has a *row buffer* that caches the last accessed row. A request that hits in the row buffer is much cheaper in terms of both latency and energy than a request that misses in the row buffer. However, bank conflicts between requests that access different rows lead to costly row buffer misses.

A naive solution to bank conflicts is to increase the number of banks. Unfortunately, as we discuss in Section 1 of our ISCA 2012 paper [66], simply adding more banks to the memory subsystem comes at significantly high costs or reduced performance regardless of the way it is done: more banks per chip, more ranks per channel, or more channels.¹

The goal in our ISCA 2012 paper [66] is to mitigate such detrimental effects of bank conflicts in a cost-effective manner. Toward that end, we make two key observations that lead to our proposed solutions.

Observation 1. A modern DRAM bank is *not* implemented as a monolithic component equipped with only a single row buffer. Implementing a DRAM bank in such a way requires very long internal wires (called *bitlines*) to connect the row buffer to all the rows in the bank, which can significantly increase the access latency. Instead, as Figure 1b shows, a bank consists of multiple *subarrays*, each with its own *local row buffer*. Subarrays within a bank share two important global structures: *i*) a *global row address decoder*, and *ii*) a *global row buffer*.

¹We refer the reader to our prior works [14, 15, 16, 17, 37, 38, 61, 62, 63, 64, 65, 66, 75, 76, 77, 78, 79, 80, 81, 115, 129, 130] for a detailed background on DRAM.

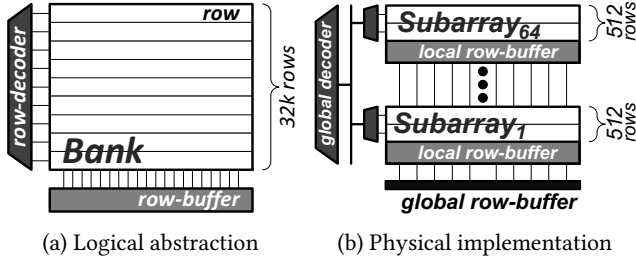


Figure 1: DRAM bank organization. Adapted from [66].

Observation 2. The latency of a bank access predominantly consists of three major components: *i*) loading a row into the local row buffer (*activation*), *ii*) accessing the data from the local row buffer (*read* or *write*), and *iii*) clearing the local row buffer (*precharging*) [14, 37, 38, 66, 76, 77, 78]. In existing DRAM banks, all three operations must be completed for one request before serving another request to a different row, even if the two rows reside in *different* subarrays. However, this does not need to be the case for two reasons. First, activation and precharging are mostly local to each subarray, which enables the opportunity to overlap these operations when they are to different subarrays. Second, if we reduce the sharing of the global structures among subarrays, we can *parallelize* the concurrent activation of different subarrays. Doing so would allow us to exploit the existence of *multiple* local row buffers across the subarrays, enabling more than just a single row to be cached for each bank and thereby increasing the row buffer hit rate.

2. Subarray-Level Parallelism

Subarray-Oblivious Baseline. Let us consider the baseline example shown in Figure 2, which presents a timeline of four memory requests being served at the same bank in a subarray-oblivious manner. This example highlights the three key problems that we discussed in Section 1. First, requests are completely serialized, even though they are to different subarrays. Second, although the write-recovery penalty is local to a subarray, it delays a subsequent request to a different subarray. Third, a request to one subarray unnecessarily evicts (i.e., *precharges*) the other subarray’s local row buffer, which must be reloaded (i.e., activated) when a future request accesses the evicted row. In this section, we describe how SALP-1, SALP-2 and MASA can take an advantage of the DRAM bank organization to enable parallel DRAM operations in a cost-effective manner.

2.1. SALP-1: Subarray-Level-Parallelism-1

We observe that precharging and activation are mostly local to a subarray. Based on this observation, we propose SALP-1, which overlaps the precharging of one subarray with the activation of another subarray. In contrast, existing systems always serialize precharging and activation to the same bank, conservatively provisioning for when they are to the same subarray. SALP-1 requires *no modifications* to

existing DRAM structure. It only requires reinterpretation of an existing timing constraint (TRP) and, potentially, the addition of a new timing constraint (which we describe in Section 5.1 of our ISCA 2012 paper [66]). Figure 3 (top) shows the timeline of the same four requests from Figure 2 when we use SALP-1 instead of our Baseline. As the timeline shows, overlapping the precharge operation reduces the overall time needed to complete the four requests.

2.2. SALP-2: Subarray-Level-Parallelism-2

While SALP-1 pipelines the precharging and activation of different subarrays, the relative ordering between the two commands is still preserved. This is because existing DRAM banks do *not* allow two subarrays to be activated *at the same time*. As a result, the write-recovery latency of an activated subarray delays not only a PRECHARGE to itself, but also a subsequent ACTIVATE to another subarray. Based on the observation that the write-recovery latency is also local to a subarray, we propose SALP-2. SALP-2 issues the ACTIVATE to another subarray *before* the PRECHARGE to the currently-activated subarray. As a result, SALP-2 can overlap the write recovery of the currently-activated subarray with the activation of another subarray, further reducing the service time compared to SALP-1 (as shown in the middle timeline of Figure 3).

However, as highlighted in the figure, SALP-2 requires two subarrays to remain activated at the same time. This is not possible in existing DRAM banks as the global row-address latch, which determines the wordline in the bank that is raised, is shared by all of the subarrays. Section 5.2 of our ISCA 2012 paper [66] discusses how to enable SALP-2 by eliminating this sharing. The key idea is to push the global address latch to each subarray, thereby creating local address latches, one per subarray.

2.3. MASA: Multitude of Activated Subarrays

Although SALP-2 allows two subarrays within a bank to be activated, it requires the controller to precharge one of them before issuing a column command (e.g., READ) to the bank. This is because when a bank receives a column command, all activated subarrays in the bank will connect their local row buffers to the global bitlines. If more than one subarray is activated, this will result in a short circuit. As a result, SALP-2 cannot allow multiple subarrays to concurrently remain activated and serve column commands.

To solve this, we propose MASA, whose key idea is to allow *multiple* subarrays to be activated at the same time, while allowing the memory controller to *designate* exactly one of the activated subarrays to drive the global bitlines during the next column command. MASA has two advantages over SALP-2. First, MASA overlaps the activation of different subarrays within a bank. Just before issuing a column command to any of the activated subarrays, the memory controller *designates* one particular subarray whose row buffer should serve the column command. Second, MASA eliminates extra ACTIVATES

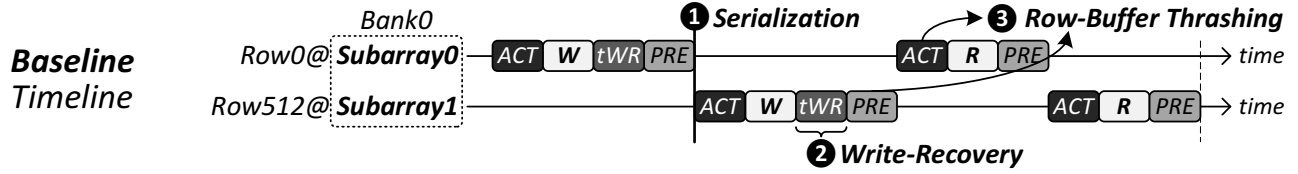


Figure 2: Timeline of four requests to two different rows in the same bank. Adapted from [66].

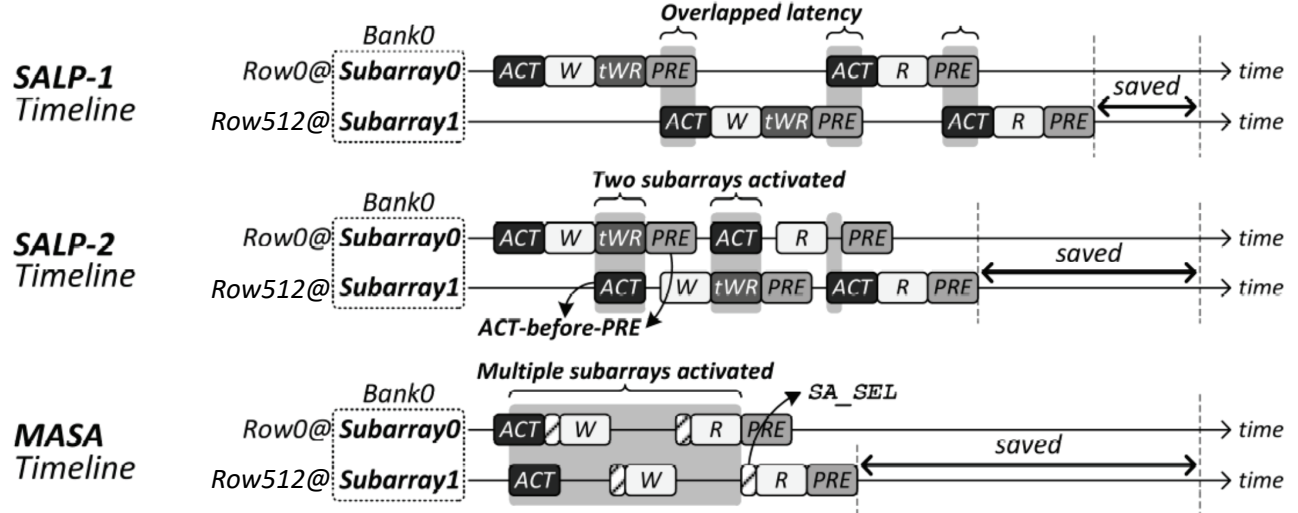


Figure 3: Timeline of four requests to two different rows in the same bank but different subarrays, using our mechanisms to exploit subarray-level parallelism. Adapted from [66].

to the same row, thereby mitigating row buffer thrashing. This is because the local row buffers of multiple subarrays can remain activated at the same time without experiencing collisions on the global bitlines. As a result, MASA further improves performance compared to SALP-2, as shown in the bottom timeline of Figure 3.

MASA: Overhead. To designate one of the multiple activated subarrays, the controller needs a new command, *SA_SEL* (*subarray-select*). In addition to the changes required by SALP-2, MASA requires a single-bit latch per subarray to denote whether a subarray is *designated* or not. According to our detailed circuit-level analysis, MASA increases the DRAM die-size by only 0.15% (due to extra latches) and the static power consumption by only $\sim 1\%$ (each additional activated subarray consumes $0.56mW$). Also, the memory controller needs less than 256 bytes to track the status of subarrays across all DRAM banks. We discuss a detailed implementation of MASA, along with its overhead, in Section 5.3 of our ISCA 2012 paper [66].

3. Experimental Methodology

We evaluate our three mechanisms for subarray-level parallelism using Ramulator [62, 124], an open-source cycle-accurate DRAM simulator that we developed which accurately models DRAM subarrays. We use Ramulator as part of a cycle-level in-house x86 multi-core simulator, whose front-end is based on Pin [85]. We calculate DRAM dynamic energy consumption by associating an energy cost with

each DRAM command, derived using Micron’s DDR3 DRAM tool [93], Rambus’ DRAM power model [123], and previously published data [150].

We evaluate SALP-1, SALP-2, and MASA on a wide variety of workloads [39, 41, 89, 146] and system configurations [45, 46, 134, 143]. The results shown in Section 4 are based on the conservative assumption that a DRAM bank exposes only 8 subarrays to be exploited by our subarray-level parallelism mechanisms, whereas in practice the number of subarrays in current DRAM banks is typically much higher (~ 64). Section 9.2 of our ISCA 2012 paper [66] shows that the performance improvement of our three mechanisms over a subarray-oblivious baseline increases with a greater number of subarrays.

For our full methodology, we refer the reader to Section 8 of our ISCA 2012 paper [66].

4. Evaluation

Figure 4 shows the performance improvement of SALP-1, SALP-2, and MASA on a system with 8 subarrays-per-bank over a subarray-oblivious baseline. The figure also shows the performance improvement of an “Ideal” scheme which is the subarray-oblivious baseline with 8 times as many banks (this represents a system where all subarrays are fully independent). The benchmarks are sorted along the x-axis by increasing memory intensity. We make two observations from the figure. First, SALP-1, SALP-2, and MASA consistently perform better than the baseline for all benchmarks.

On average, they improve the average performance by 6.6%, 13.4%, and 16.7%, respectively. Second, MASA captures most of the benefits of “Ideal,” which improves performance by 19.6% compared to baseline.

The difference in performance improvement across benchmarks can be explained by a combination of three factors related to the benchmarks’ individual memory access behavior. First, subarray-level parallelism in general is most beneficial for memory-intensive benchmarks that frequently access memory (e.g., the benchmarks located towards the right of Figure 4). By increasing the memory throughput for such applications, subarray-level parallelism significantly alleviates their memory bottleneck. The average memory intensity of the applications that gain >5% performance with SALP-1 is 18.4 MPKI (last-level cache misses per kilo-instruction), compared to 1.14 MPKI for the other applications.

Second, the advantage of SALP-2 is large for applications that are write-intensive (i.e., those with the most write misses per kilo-instruction, or WMPKI). For such applications, SALP-2 can overlap the long write-recovery latency with the activation of a subsequent access. In Figure 4, the three applications that improve more than 38% with SALP-2 are among both the most memory-intensive (>25 MPKI) and the most write-intensive (>15 WMPKI).

Third, MASA is beneficial for applications that experience frequent bank conflicts. For such applications, MASA parallelizes accesses to different subarrays by concurrently activating multiple subarrays (ACTIVATE) and allowing the application to switch between the activated subarrays at low cost (SA_SEL). Therefore, the subarray-level parallelism offered by MASA can be gauged by the SA_SEL-to-ACTIVATE ratio. For the nine applications that benefit more than 30% from MASA, on average, one SA_SEL was issued for every two ACTIVATES, compared to one-in-seventeen for all other applications. For a few benchmarks, MASA performs slightly worse than SALP-2. This is because the baseline scheduling algorithm used with MASA tries to overlap as many ACTIVATES as possible, and in the process inadvertently delays the column command of the most critical request. This delay to the most critical request slightly degrades performance for these benchmarks.²

Energy Efficiency. We focus on the energy efficiency of MASA. MASA utilizes multiple local row buffers across subarrays and increases the chance that an access will hit in a local row buffer. Specifically, MASA increases the row buffer hit rate by an average of 12.8% across 32 benchmarks. A row buffer hit not only has a lower access latency, but also consumes less energy, since it does not require the power-hungry operations of activation and, to a lesser degree, precharging. Consequently, MASA reduces the dynamic energy consumption by 18.6% as shown in Figure 5.

Our ISCA 2012 paper [66] provides a detailed evaluation of SALP-1, SALP-2, and MASA, including:

- Sensitivity studies to (1) the number of channels (1–8), ranks (1–8), banks (8–64), and subarrays per bank (1–128) in the memory system; (2) the mapping policy (row-/line-interleaved); and (3) an open-row or closed-policy (Sections 9.2 and 9.3 of [66]).
- Multi-core results using an application-aware memory scheduling algorithm, where we show significant performance improvements (Section 9.3 of [66]).
- An analysis of the power and area overhead at both the DRAM chip and the memory controller (Section 6 of [66]).

5. Related Work

To our knowledge, our ISCA 2012 paper [66] is the first to exploit the existence of subarrays within a DRAM bank and enable their parallel operation in a cost-effective manner. We propose three schemes that exploit the existence of subarrays within DRAM banks to mitigate the negative effects of bank conflicts. Related works propose increasing the performance and energy-efficiency of DRAM through approaches such as DRAM module reorganization, changes to DRAM chip design, and memory controller optimizations. We briefly discuss these works here.

DRAM Module Reorganization. Several prior works [3, 4, 151, 164] partition a DRAM rank and the DRAM data bus into multiple rank subsets, each of which can be operated independently. While these techniques increase parallelism, they reduce the width of the data bus of each rank subset, leading to longer latencies to transfer a 64 byte cache line. Furthermore, having many rank subsets requires a correspondingly large number of DRAM chips to compose a DRAM rank, an assumption that does not hold in mobile DRAM systems where a rank may consist of as few as two chips [95]. Unlike these works, our mechanisms increase memory-level parallelism [72, 100, 101, 105, 107, 108, 120] without increasing memory latency or the number of DRAM chips.

Changes to DRAM Design. Cached DRAM organizations, which have been widely proposed [25, 36, 40, 44, 56, 110, 125, 152, 161], augment DRAM chips with an additional SRAM cache that can store recently accessed data in order to reduce memory access latency. However, these proposals increase the chip area and design complexity of DRAM designs. Furthermore, cached DRAM provides parallelism only when accesses *hit* in the SRAM cache, while serializing cache misses that access the same DRAM bank. Our schemes parallelize DRAM bank accesses while incurring significantly lower area and logic complexity.

Fujitsu’s FCRAM [126] and Micron’s RLDRAM [57] propose to implement shorter local bitlines (i.e., fewer cells per bitline) that are quickly drivable due to their lower capacitance in order to reduce DRAM latency. However, this significantly increases the DRAM die size (30–40% for FCRAM, 40–80% for RLDRAM) because the large area of sense-amplifiers is amortized over a smaller number of cells. Hybrid memory systems can reduce the die size overhead by using a small

²For one benchmark, MASA performs slightly better than “Ideal” due to interactions with the scheduler.

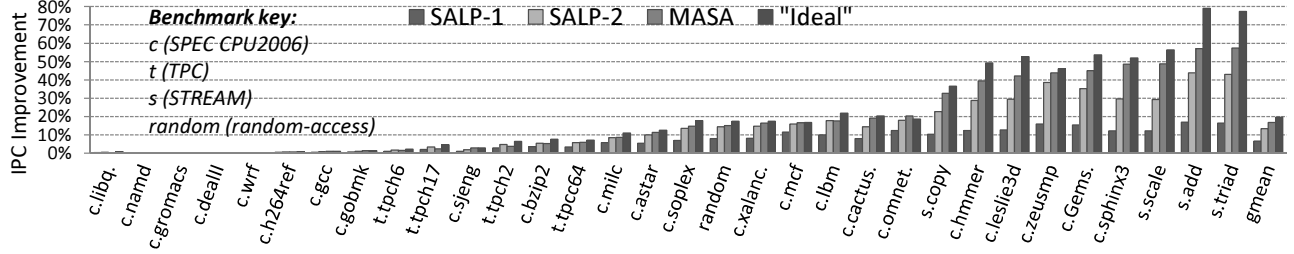


Figure 4: IPC improvement of SALP-1, SALP-2, MASA, and an ideal mechanism over the subarray-oblivious baseline. Reproduced from [66].

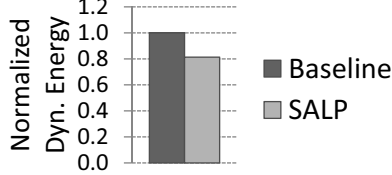


Figure 5: Dynamic DRAM energy consumption for MASA.

amount of FCRAM [126] or RLDRAM [57] in conjunction with conventional DRAM and managing which subset of the data resides in FCRAM/RLDRAM at any given time to lower the latency of memory accesses.

A patent by Qimonda [113] proposes the high-level notion of separately addressable sub-banks, but lacks concrete mechanisms for exploiting the independence between sub-banks. Yamauchi et al. propose the Hierarchical Multi-Bank (HMB) [154], which parallelizes accesses to different subarrays in a fine-grained manner. However, this scheme adds complex logic to all subarrays.

Udipi et al. [147] propose two techniques (SBA and SSA) to lower DRAM power. In SBA, global wordlines are segmented and controlled separately so that tiles in the horizontal direction are not activated in lockstep, but selectively. However, this increases DRAM chip area by 12-100% [147]. SSA combines SBA with chip-granularity rank-subsetting to achieve even higher energy savings. Both SBA and SSA increase DRAM latency, more significantly so for SSA (due to rank-subsetting).

When transitioning from serving a write request to serving a read request, and vice versa [18, 73, 137], a DRAM chip experiences bubbles in the data bus, called the *bus-turnaround penalty* (t_{WTR} and t_{RTW}). During the bus turnaround penalty, Chatterjee et al. [18] propose to internally “prefetch” data for subsequent read requests into extra registers that are added to the DRAM chip.

Other works propose new DRAM designs that are capable of reducing memory latency of conventional DRAM [3, 4, 14, 16, 19, 36, 40, 44, 56, 75, 76, 77, 78, 79, 86, 94, 112, 118, 126, 133, 135, 151, 164] as well as non-volatile memory [68, 69, 70, 71, 90, 91, 121, 122, 155]. Previous works on bulk data transfer [13, 16, 33, 34, 47, 51, 53, 84, 127, 129, 158, 163] and in-memory computation [1, 2, 5, 9, 11, 12, 23, 26, 27, 28, 29, 30, 32, 35, 42, 43, 55, 60, 67, 88, 114, 116, 117, 119, 128, 130, 131, 132, 136, 144, 157] can be used improve DRAM bandwidth utilization and lower

the number of costly data movements between CPU cores and DRAM. All these works can benefit from SALP as the underlying memory substrate.

Memory Controller Optimizations. To reduce bank conflicts and increase row buffer locality, Zhang et al. [160] propose to randomize the bank address of memory requests by XOR hashing. Sudan et al. [142] propose to improve row buffer locality by placing frequently-referenced data from different rows together in the same row buffer. Both proposals can be combined with our mechanisms to further improve parallelism and row buffer locality.

Prior works propose memory scheduling algorithms for CPUs (e.g., [24, 31, 48, 58, 59, 64, 65, 72, 73, 74, 82, 96, 97, 98, 99, 106, 107, 111, 137, 138, 139, 140, 141, 153, 162]), GPUs (e.g., [7, 8, 20, 50, 52]), and other systems (e.g., [148, 149, 162]) that prioritize certain favorable requests in the memory controller to improve system performance and/or fairness. Subarrays expose more parallelism to the memory controller, increasing the controller’s flexibility to schedule requests. Our subarray-level parallelism mechanisms can be combined with many of these schedulers to provide increased performance benefits. Enabling higher benefit from SALP by designing SALP-aware memory scheduling algorithms is a promising open research topic.

6. Significance and Long-Term Impact

We believe SALP will have long-term impact because: *i*) it tackles a critical problem, bank conflicts and memory parallelism, whose importance will increase in the future; and *ii*) the memory substrate it provides can further be leveraged to enable other novel optimizations in the memory subsystem. In fact, as Section 6.2 shows, there has been a significant amount of work that built upon our ISCA 2012 paper in the past six years.

6.1. Trends and Opportunities in Favor of SALP

Worsening Bank Conflicts. Future many-core systems with large numbers of cores and accelerators (e.g., bandwidth-hungry GPUs) will exert increasingly larger amount of pressure on the memory subsystem. On the other hand, naively adding more DRAM banks is difficult without incurring high costs, high energy or reduced performance. Therefore, as more and more memory requests contend to access a limited

er of banks, bank conflicts will occur with increasing likelihood and severity. SALP is a cost-effective mechanism to alleviate the bank conflict problem by exploiting the existing subarrays in DRAM at low cost.

Challenges in DRAM Scaling. DRAM process scaling is becoming more difficult due to increased manufacturing complexity/cost and reduced cell reliability [6, 49, 54, 63, 102, 103, 104, 109]. As a result, it is critical to examine alternative ways of improving memory performance while still maintaining low cost. SALP is a new cost-effective DRAM design whose advantages are mostly orthogonal to the advantages of DRAM process scaling. Therefore, SALP can further improve the performance and the energy-efficiency of future DRAM. In fact, a recent industry proposal to enhance the DDR standard incorporates one of our SALP mechanisms [54]. This work by Samsung and Intel quantitatively shows that SALP is an effective mechanism to tolerate increasing write latencies in DRAM, corroborating the results in our ISCA 2012 paper on SALP-2.

A Building Block for New Optimizations. SALP enables new DRAM optimizations that were not possible before. We discuss three potential examples. First, exploiting subarray-level parallelism can potentially mitigate DRAM unavailability during refresh by parallelizing refreshes in one subarray with accesses to another subarray within the same bank. Work by Chang et al. [15], which builds on our ISCA 2012 paper, shows that such parallelization can eliminate most of the performance overhead of refresh. Second, subarrays provide an additional degree of freedom in mapping the physical address space onto different levels of the DRAM hierarchy (channels, ranks, banks, subarrays, rows, columns). Thus, they enable more flexibility in performance and energy optimization via data mapping. Third, DRAM can be divided among different applications (to provide quality-of-service) at the finer-grained partitions of subarrays that are less vulnerable to capacity and bandwidth fragmentation. As we discuss, some research has explored these approaches (also see Section 6.2). We expect even more future research will tap into these and other opportunities that can use our proposed SALP substrate as a building block for other optimizations.

Widely Applicable Substrate. SALP is a general-purpose substrate that is also applicable to embedded DRAM (eDRAM) [10] and 3D die-stacked DRAM (3D-DRAM), both of which consist of subarrays [75, 83]. For example, eDRAM is known to be vulnerable to the write-recovery penalty [22], since it is typically used as the last-level cache and thus exposed to higher amounts of write traffic. SALP can increase the availability of eDRAM by hiding the write-recovery penalty. In addition, SALP may be applied to future emerging memory technologies as long as their banks are organized hierarchically [69, 92], similar to how a DRAM bank consists of subarrays.

New Research Opportunities. SALP creates new opportunities for exploiting and enhancing the parallelism and the locality of the memory subsystem.

- *Enhancing Memory-Level Parallelism.* To tolerate the long latency of DRAM, computer architects often design mechanisms that perform multiple memory requests in a concurrent manner [72, 100, 101, 105, 107, 108, 120, 145]. Such efforts may become ineffective when requests access the same DRAM bank and, as a consequence, are not actually served in parallel [107]. SALP, on the other hand, parallelizes requests to different subarrays *within the same bank*. In this regard, we believe SALP not only enhances previous approaches to memory-level parallelism, but also creates opportunities for developing new techniques that preserve memory-level parallelism in a subarray-aware manner.
- *Enhancing Memory Locality.* Memory access patterns that exhibit high locality benefit greatly from a DRAM bank's row buffer where the last accessed row is cached (4–8kB). While a DRAM bank has multiple row buffers across multiple subarrays, an existing DRAM system exposes only one row buffer at a time in a bank and, as a result, is prone to row buffer thrashing. In contrast, SALP allows a DRAM bank to utilize multiple row buffers concurrently. This enables the opportunity for new techniques that can take advantage of the multiple row buffers, whether they be for streaming/strided accesses (demand or prefetch), vector processing, or GPUs.

6.2. Works Building on SALP

The introduction of the notion of subarrays and their microarchitecture has enabled the use of the subarrays in many works. These include RowClone [129], TL-DRAM [78], DSARP [15], DIVA-DRAM [76], LISA [16], ChargeCache [37], Multiple Clone Row DRAM [21], Ambit [128, 130], ERUCA [87], and other works on improving DRAM [84, 135, 156, 159]. Some of these works exploit subarray level parallelism, e.g., DSARP [15] reduces the overhead of a DRAM refresh by decoupling independent subarrays from the subarray that is being refreshed. This decoupling allows DRAM to service memory accesses while a subarray is being refreshed. Others make changes to subarrays to improve an aspect, e.g., TL-DRAM [78] creates two different latency regions in a subarray to improve DRAM latency at low cost.

7. Conclusion

Our ISCA 2012 paper [66] introduces three new mechanisms that exploit the existence of subarrays within a DRAM bank to mitigate the performance impact of bank conflicts. Our mechanisms are built on the key observation that subarrays within a DRAM bank operate largely independently and have their own row buffers. Hence, the latencies of accesses to different subarrays within the same bank can potentially be overlapped to a large degree. Our three mechanisms take

advantage of this fact and progressively increase the independence of operation of subarrays by making small modifications to the DRAM chip. Our most sophisticated scheme, MASA, enables *i*) multiple subarrays to be accessed in parallel, and *ii*) multiple row buffers to remain activated at the same time in different subarrays, thereby improving both memory-level parallelism and row buffer locality. We show that our schemes significantly improve system performance on both single-core and multi-core systems on a variety of workloads while incurring little ($<0.15\%$) or no area overhead in the DRAM chip. Our techniques can also improve memory energy efficiency.

We conclude that exploiting subarray-level parallelism in a DRAM bank can be a promising and cost-effective method for overcoming the negative effects of DRAM bank conflicts, without paying the large cost of increasing the number of banks in the DRAM system. Significant recent work has built upon our ISCA 2012 paper, and we expect many other new works can exploit the new substrate we have enabled to achieve even bigger goals and higher benefits.

Acknowledgments

We thank Rachata Ausavarungnirun and Saugata Ghose for their dedicated effort in the preparation of this article. Many thanks to Uksong Kang, Hak-soo Yu, Churoo Park, Jung-Bae Lee, and Joo Sun Choi from Samsung for their helpful comments. We thank the anonymous reviewers for their feedback. We gratefully acknowledge members of the SAFARI group for feedback and for the stimulating intellectual environment they provide. We acknowledge the generous support of AMD, Intel, Oracle, and Samsung. This research was also partially supported by grants from NSF (CAREER Award CCF-0953246), GSRC, and Intel ARO Memory Hierarchy Program.

References

- [1] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A Scalable Processing-in-memory Accelerator for Parallel Graph Processing," in *ISCA*, 2015.
- [2] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "PIM-enabled Instructions: A Low-overhead, Locality-aware Processing-in-memory Architecture," in *ISCA*, 2015.
- [3] J. H. Ahn, J. Leverich, R. Schreiber, and N. P. Jouppi, "Multicore DIMM: an Energy Efficient Memory Module with Independently Controlled DRAMs," *IEEE CAL*, 2009.
- [4] J. H. Ahn, N. P. Jouppi, C. Kozyrakis, J. Leverich, and R. S. Schreiber, "Improving System Energy Efficiency with Memory Rank Subsetting," *TACO*, 2012.
- [5] B. Akin, F. Franchetti, and J. C. Hoe, "Data Reorganization in Memory Using 3D-stacked DRAM," in *ISCA*, 2015.
- [6] G. Atwood, "Current and Emerging Memory Technology Landscape (Micron)," in *Flash Memory Summit*, 2011.
- [7] R. Ausavarungnirun, K. K. Chang, L. Subramanian, G. Loh, and O. Mutlu, "Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems," in *ISCA*, 2012.
- [8] R. Ausavarungnirun, S. Ghose, O. Kayiran, G. H. Loh, C. R. Das, M. T. Kandemir, and O. Mutlu, "Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance," in *PACT*, 2015.
- [9] O. O. Babarinsa and S. Idreos, "JAFAR: Near-Data Processing for Databases," in *SIGMOD*, 2015.
- [10] J. Barth, D. Plass, E. Nelson, C. Hwang, G. Fredeman, M. Sperling, A. Mathews, T. Kirihaata, W. R. Rehr, K. Nair, and N. Caon, "A 45 nm SOI Embedded DRAM Macro for the POWER Processor 32 MByte On-Chip L3 Cache," *JSSC*, 2011.
- [11] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, and O. Mutlu, "Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks," in *ASPLOS*, 2018.
- [12] A. Boroumand, S. Ghose, B. Lucia, K. Hsieh, K. Malladi, H. Zheng, and O. Mutlu, "LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory," *IEEE CAL*, 2016.
- [13] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama, "Impulse: Building a Smarter Memory Controller," in *HPCA*, 1999.
- [14] K. K. Chang, A. Kashyap, H. Hassan, S. Ghose, K. Hsieh, D. Lee, T. Li, G. Pehimenco, S. Khan, and O. Mutlu, "Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization," in *SIGMETRICS*, 2016.
- [15] K. K. Chang, D. Lee, Z. Chishti, A. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu, "Improving DRAM Performance by Parallelizing Refreshes with Accesses," in *HPCA*, 2014.
- [16] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu, "Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM," in *HPCA*, 2016.
- [17] K. K. Chang, A. G. Yaglikci, S. Ghose, A. Agrawal, N. Chatterjee, A. Kashyap, D. Lee, M. O'Connor, H. Hassan, and O. Mutlu, "Understanding Reduced-Voltage Operation in Modern DRAM Devices: Experimental Characterization, Analysis, and Mechanisms," in *SIGMETRICS*, 2017.
- [18] N. Chatterjee *et al.*, "Staged reads: Mitigating the impact of DRAM writes on DRAM reads," in *HPCA*, 2012.
- [19] N. Chatterjee, M. Shevgoor, R. Balasubramanian, A. Davis, Z. Fang, R. Illikkal, and R. Iyer, "Leveraging Heterogeneity in DRAM Main Memories to Accelerate Critical Word Access," in *MICRO*, 2012.
- [20] N. Chatterjee, M. O'Connor, G. H. Loh, N. Jayasena, and R. Balasubramanian, "Managing DRAM Latency Divergence in Irregular GPGPU Applications," in *SC*, 2014.
- [21] J. Choi, W. Shin, J. Jang, J. Suh, Y. Kwon, Y. Moon, and L.-S. Kim, "Multiple Clone Row DRAM: A Low Latency and Area Optimized DRAM," in *ISCA*, 2015.
- [22] B. W. Curran *et al.*, "The zEnterprise 196 System and Microprocessor," *IEEE Micro*, Mar 2011.
- [23] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, and G. Daglikoca, "The Architecture of the DIVA Processing-in-memory Chip," in *ICS*, 2002.
- [24] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt, "Parallel Application Memory Scheduling," in *MICRO*, 2011.
- [25] Enhanced Memory Systems, "Enhanced SDRAM SM2604," 2002.
- [26] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, "NDA: Near-DRAM Acceleration Architecture Leveraging Commodity DRAM Devices and Standard Memory Modules," in *HPCA*, 2015.
- [27] B. B. Fraguela, J. Renau, P. Feautrier, D. Padua, and J. Torrellas, "Programming the FlexRAM Parallel Intelligent Memory System," in *PPoPP*, 2003.
- [28] M. Gao, G. Ayers, and C. Kozyrakis, "Practical Near-Data Processing for In-Memory Analytics Frameworks," in *PACT*, 2015.
- [29] M. Gao and C. Kozyrakis, "HRL: Efficient and Flexible Reconfigurable Logic for Near-Data Processing," in *HPCA*, 2016.
- [30] S. Ghose, K. Hsieh, A. Boroumand, R. Ausavarungnirun, and O. Mutlu, "The Processing-in-Memory Paradigm: Mechanisms to Enable Adoption," *arXiv [cs.AR]*, 2018.
- [31] S. Ghose, H. Lee, and J. F. Martinez, "Improving Memory Scheduling via Processor-side Load Criticality Information," in *ISCA*, 2013.
- [32] M. Gokhale, B. Holmes, and K. Iobst, "Processing in Memory: the Terasys Massively Parallel PIM Array," *Computer*, 1995.
- [33] M. Gschwind, "Chip Multiprocessing and the Cell Broadband Engine," in *CF*, 2006.
- [34] J. Gummara, M. Erez, J. Coburn, M. Rosenblum, and W. J. Dally, "Architectural Support for the Stream Execution Model on General-Purpose Processors," in *PACT*, 2007.
- [35] Q. Guo, N. Alachiotis, B. Akin, F. Sadi, G. Xu, T.-M. Low, L. Pileggi, J. C. Hoe, and F. Franchetti, "3D-Stacked Memory-Side Acceleration: Accelerator and System Design," in *WoNDP*, 2014.
- [36] C. A. Hart, "CDRAM in a Unified Memory Architecture," in *Compcon*, 1994.
- [37] H. Hassan, G. Pehimenco, N. Vijaykumar, V. Seshadri, D. Lee, O. Ergin, and O. Mutlu, "ChargeCache: Reducing DRAM Latency by Exploiting Row Access Locality," in *HPCA*, 2016.
- [38] H. Hassan, N. Vijaykumar, S. Khan, S. Ghose, K. K. Chang, G. Pehimenco, D. Lee, O. Ergin, and O. Mutlu, "SoftMC: A Flexible and Practical Open-source Infrastructure for Enabling Experimental DRAM Studies," in *HPCA*, 2017.
- [39] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *CAN*, 2006.
- [40] H. Hidaka, Y. Matsuda, M. Asakura, and K. Fujishima, "The Cache DRAM Architecture," *IEEE Micro*, 1990.
- [41] HPC Challenge, "GUPS," <http://icl.cs.utk.edu/projectsfiles/hpcc/RandomAccess/>.
- [42] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu, "Accelerating Pointer Chasing in 3D-stacked Memory: Challenges, Mechanisms, Evaluation," in *ICCD*, 2016.
- [43] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler, "Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems," in *ISCA*, 2016.

- [44] W.-C. Hsu and J. E. Smith, "Performance of Cached DRAM Organizations in Vector Supercomputers," in *ISCA*, 1993.
- [45] Intel, "2nd Gen. Intel Core Processor Family Desktop Datasheet," 2011.
- [46] Intel, "Intel Core Desktop Processor Series Datasheet," 2011.
- [47] Intel Corp., "Intel® I/O Acceleration Technology," <http://www.intel.com/content/www/us/en/wireless-network/accel-technology.html>.
- [48] E. Ipek, O. Mutlu, J. F. Martinez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," in *ISCA*, 2008.
- [49] ITRS, "International Technology Roadmap for Semiconductors," 2011.
- [50] M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver, "A QoS-Aware Memory Controller for Dynamically Balancing GPU and CPU Bandwidth Use in an MPSoC," in *DAC*, 2012.
- [51] X. Jiang, Y. Solihin, L. Zhao, and R. Iyer, "Architecture Support for Improving Bulk Memory Copying and Initialization Performance," in *PACT*, 2009.
- [52] A. Jog, O. Kayiran, A. Pattnaik, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Exploiting Core Criticality for Enhanced GPU Performance," in *SIGMETRICS*, 2016.
- [53] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the Cell Multiprocessor," *IBM JRD*, 2005.
- [54] U. Kang, H.-S. Yu, C. Park, H. Zheng, J. Halbert, K. Bains, S. Jang, and J. Choi, "Co-Architecting Controllers and DRAM to Enhance DRAM Process Scaling," in *The Memory Forum*, 2014.
- [55] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas, "FlexRAM: Toward an Advanced Intelligent Memory System," in *ICCD*, 2009.
- [56] G. Kedem and R. P. Koganti, "WCDRAM: A Fully Associative Integrated Cached-DRAM with Wide Cache Lines," Duke Univ. Dept. of Computer Science, Tech. Rep. CS-1997-03, 1997.
- [57] B. Keeth et al., *DRAM Circuit Design. Fundamental and High-Speed Topics*. Wiley-IEEE Press, 2007.
- [58] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, "Bounding Memory Interference Delay in COTS-based Multi-core Systems," in *RTAS*, 2014.
- [59] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, "Bounding and Reducing Memory Interference in COTS-based Multi-core Systems," *RTS*, 2016.
- [60] J. S. Kim, D. Senol, H. Xin, D. Lee, S. Ghose, M. Alser, H. Hassan, O. Ergin, C. Alkan, and O. Mutlu, "GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-in-Memory Technologies," *BMC Genomics*, 2018.
- [61] J. S. Kim, M. Patel, H. Hassan, and O. Mutlu, "The DRAM Latency PUF: Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency-Reliability Tradeoff in Modern DRAM Devices," in *HPCA*, 2018.
- [62] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," *IEEE CAL*, 2015.
- [63] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," in *ISCA*, 2014.
- [64] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers," in *HPCA*, 2010.
- [65] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *MICRO*, 2010.
- [66] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM," in *ISCA*, 2012.
- [67] P. M. Kogge, "EXECUBE-A New Architecture for Scaleable MPPs," in *ICPP*, 1994.
- [68] E. Kultursay, M. Kandemir, A. Sivasubramanian, and O. Mutlu, "Evaluating STT-RAM as an energy-efficient main memory alternative," in *ISPASS*, 2013.
- [69] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting Phase Change Memory as a Scalable DRAM Alternative," in *ISCA*, 2009.
- [70] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Phase Change Memory Architecture and the Quest for Scalability," *CACM*, 2010.
- [71] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-Change Technology and the Future of Main Memory," *IEEE Micro*, 2010.
- [72] C. J. Lee, V. Narasiman, O. Mutlu, and Y. N. Patt, "Improving Memory Bank-Level Parallelism in the Presence of Prefetching," in *MICRO*, 2009.
- [73] C. J. Lee, E. Ebrahimi, V. Narasiman, O. Mutlu, and Y. N. Patt, "DRAM-Aware Last-Level Cache Writeback: Reducing Write-Caused Interference in Memory Systems," Univ. of Texas at Austin, High Performance Systems Group, Tech. Rep. TR-HPS-2010-002, 2010.
- [74] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt, "Prefetch-aware DRAM Controllers," in *MICRO*, 2008.
- [75] D. Lee, S. Ghose, G. Pekhimenko, S. Khan, and O. Mutlu, "Simultaneous Multi-layer Access: Improving 3D-stacked Memory Bandwidth at Low Cost," *TACO*, 2016.
- [76] D. Lee, S. Khan, L. Subramanian, S. Ghose, R. Ausavarungnirun, G. Pekhimenko, V. Seshadri, and O. Mutlu, "Design-Induced Latency Variation in Modern DRAM Chips: Characterization, Analysis, and Latency Reduction Mechanisms," in *SIGMETRICS*, 2017.
- [77] D. Lee, Y. Kim, G. Pekhimenko, S. Khan, V. Seshadri, K. K. Chang, and O. Mutlu, "Adaptive-latency DRAM: Optimizing DRAM Timing for the Common-case," in *HPCA*, 2015.
- [78] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu, "Tiered-latency DRAM: A Low Latency and Low Cost DRAM Architecture," in *HPCA*, 2013.
- [79] D. Lee, L. Subramanian, R. Ausavarungnirun, J. Choi, and O. Mutlu, "Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM," in *PACT*, 2015.
- [80] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu, "An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms," in *ISCA*, 2013.
- [81] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, "RAIDR: Retention-Aware Intelligent DRAM Refresh," in *ISCA*, 2012.
- [82] W. Liu, P. Huang, T. Kun, T. Lu, K. Zhou, C. Li, and X. He, "LAMS: A Latency-aware Memory Scheduling Policy for Modern DRAM Systems," in *IPCCC*, 2016.
- [83] G. H. Loh, "3D-stacked Memory Architectures for Multi-core Processors," in *ISCA*, 2008.
- [84] S.-L. Lu, Y.-C. Lin, and C.-L. Yang, "Improving DRAM Latency with Dynamic Asymmetric Subarray," in *MICRO*, 2015.
- [85] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.
- [86] Y. Luo, S. Govindan, B. Sharma, M. Santaniello, J. Meza, A. Kansal, J. Liu, B. Khes-sib, K. Vaid, and O. Mutlu, "Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-Reliability Memory," in *DSN*, 2014.
- [87] S. Lym, H. Ha, Y. Kwon, C. Chang, J. Kim, and M. Erez, "ERUCA: Efficient DRAM Resource Utilization and Resource Conflict Avoidance for Memory System Parallelism," in *HPCA*, 2018.
- [88] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, "Smart Memories: A Modular Reconfigurable Architecture," in *ISCA*, 2000.
- [89] J. D. McCalpin, "STREAM Benchmark," <http://www.streambench.org/>.
- [90] J. Meza, Y. Luo, S. Khan, J. Zhao, Y. Xie, and O. Mutlu, "A Case for Efficient Hardware/Software Cooperative Management of Storage and Memory," in *WEED*, 2013.
- [91] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan, "Enabling Efficient and Scalable Hybrid Memories Using Fine-Granularity DRAM Cache Management," *IEEE CAL*, 2012.
- [92] J. Meza, J. Li, and O. Mutlu, "A Case for Small Row Buffers in Non-volatile Main Memories," in *ICCD*, 2012.
- [93] Micron Technology, Inc., "DDR3 SDRAM System-Power Calculator," 2010.
- [94] Micron Technology, Inc., "576Mb: x18, x36 RLD3RAM3," 2011.
- [95] Micron Technology, Inc., "2Gb: x16, x32 Mobile LPDDR2 SDRAM," 2012.
- [96] T. Moscibroda and O. Mutlu, "Memory Performance Attacks: Denial of Memory Service in Multi-core Systems," in *USENIX Security*, 2007.
- [97] T. Moscibroda and O. Mutlu, "Distributed Order Scheduling and Its Application to Multi-core DRAM Controllers," in *PODC*, 2008.
- [98] J. Mukundan and J. F. Martinez, "MORSE: Multi-objective Reconfigurable Self-optimizing Memory Scheduler," in *HPCA*, 2012.
- [99] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, "Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning," in *MICRO*, 2011.
- [100] O. Mutlu, H. Kim, and Y. N. Patt, "Address-value Delta (AVD) Prediction: Increasing the Effectiveness of Runahead Execution by Exploiting Regular Memory Allocation Patterns," in *MICRO*, 2005.
- [101] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An effective alternative to large instruction windows," *IEEE Micro*, 2003.
- [102] O. Mutlu, "Memory Scaling: A Systems Architecture Perspective," *IMW*, 2013.
- [103] O. Mutlu, "Memory Scaling: A Systems Architecture Perspective," in *MEMCON*, 2013.
- [104] O. Mutlu, "The RowHammer Problem and Other Issues We May Face as Memory Becomes Denser," in *DATE*, 2017.
- [105] O. Mutlu, H. Kim, and Y. N. Patt, "Techniques for Efficient Processing in Runahead Execution Engines," in *ISCA*, 2005.
- [106] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *MICRO*, 2007.
- [107] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems," in *ISCA*, 2008.
- [108] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors," in *HPCA*, 2003.
- [109] O. Mutlu and L. Subramanian, "Research Problems and Opportunities in Memory Systems," *SUPERFRI*, 2014.
- [110] NEC, "Virtual Channel SDRAM uPD4565421," 1999.
- [111] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith, "Fair queuing memory systems," in *MICRO*, 2006.
- [112] S. O, Y. H. Son, N. S. Kim, and J. H. Ahn, "Row-Buffer Decoupling: A Case for Low-Latency DRAM Microarchitecture," in *ISCA*, 2014.
- [113] J.-H. Oh, "Semiconductor memory having a bank with sub-banks," U.S. Patent 7,782,703, 2010.
- [114] M. Oskin, F. T. Chong, and T. Sherwood, "Active Pages: A Computation Model for Intelligent Memory," in *ISCA*, 1998.
- [115] M. Patel, J. Kim, and O. Mutlu, "The Reach Profiler (REAPER): Enabling the Mitigation of DRAM Retention Failures via Profiling at Aggressive Conditions," in

- ISCA, 2017.
- [116] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A Case for Intelligent RAM," *IEEE Micro*, 1997.
 - [117] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das, "Scheduling Techniques for GPU Architectures with Processing-In-Memory Capabilities," in *PACT*, 2016.
 - [118] S. Phadke and S. Narayanasamy, "MLP Aware Heterogeneous Memory System," in *DATE*, 2011.
 - [119] S. H. Pugsley, J. Jests, H. Zhang, R. Balasubramanian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "NDC: Analyzing the Impact of 3D-stacked Memory+logic Devices on MapReduce Workloads," in *ISPASS*, 2014.
 - [120] M. K. Qureshi, D. Lynch, O. Mutlu, and Y. Patt, "A Case for MLP-Aware Cache Replacement," in *ISCA*, 2006.
 - [121] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing Lifetime and Security of PCM-based Main Memory with Start-gap Wear Leveling," in *MICRO*, 2009.
 - [122] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable High Performance Main Memory System Using Phase-change Memory Technology," in *ISCA*, 2009.
 - [123] Rambus, "DRAM Power Model," 2010.
 - [124] SAFARI Research Group, "Ramulator – GitHub Repository," <https://github.com/CMU-SAFARI/ramulator>.
 - [125] R. H. Sartore, K. J. Mobley, D. G. Carrigan, and O. F. Jones, "Enhanced DRAM with embedded registers," U.S. Patent 5,887,272, 1999.
 - [126] Y. Sato, T. Suzuki, T. Aikawa, S. Fujioka, W. Fujieda, H. Kobayashi, H. Ikeda, T. Nagasawa, A. Funyu, Y. Fuji, K. Kawasaki, M. Yamazaki, and M. Taguchi, "Fast cycle RAM (FCRAM): A 20-ns Random Row Access, Pipe-Lined Operating DRAM," in *VLSI*, 1998.
 - [127] S.-Y. Seo, "Methods of Copying a Page in a Memory Device and Methods of Managing Pages in a Memory System," U.S. Patent Application 20140185395, 2014.
 - [128] V. Seshadri, K. Hsieh, A. Boroumand, D. Lee, M. Kozuch, O. Mutlu, P. Gibbons, and T. Mowry, "Fast Bulk Bitwise AND and OR in DRAM," *IEEE CAL*, 2015.
 - [129] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization," in *ISCA*, 2013.
 - [130] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-memory Accelerator for Bulk Bitwise Operations using Commodity DRAM Technology," in *MICRO*, 2017.
 - [131] V. Seshadri, T. Mullins, A. Boroumand, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Gather-Scatter DRAM: In-DRAM Address Translation to Improve the Spatial Locality of Non-Unit Strided Accesses," in *MICRO*, 2015.
 - [132] V. Seshadri and O. Mutlu, "Simple Operations in Memory to Reduce Data Movement," in *Advances in Computers, Volume 106*, 2017.
 - [133] W. Shin, J. Yang, J. Choi, and L.-S. Kim, "NUAT: A Non-Uniform Access Time Memory Controller," in *HPCA*, 2014.
 - [134] B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie *et al.*, "IBM POWER7 multicore server processor," *IBM Journal Res. Dev.*, May, 2011.
 - [135] Y. H. Son, S. O. Y. Ro, J. W. Lee, and J. H. Ahn, "Reducing Memory Access Latency with Asymmetric DRAM Bank Organizations," in *ISCA*, 2013.
 - [136] H. S. Stone, "A Logic-in-Memory Computer," *IEEE TC*, vol. C-19, no. 1, pp. 73–78, 1970.
 - [137] 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*, 2010.
 - [138] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, "BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling," in *IEEE TPDS*, 2016.
 - [139] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, "The Blacklisting Memory Scheduler: Achieving high performance and fairness at low cost," in *ICCD*, 2014.
 - [140] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu, "The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-application Interference at Shared Caches and Main Memory," in *MICRO*, 2015.
 - [141] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu, "MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems," in *HPCA*, 2013.
 - [142] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramanian, and A. Davis, "Micro-pages: Increasing DRAM efficiency with locality-aware data placement," in *ASPLOS*, 2010.
 - [143] Sun Microsystems, "OpenSPARC T1 microarch. specification," 2006.
 - [144] Z. Sura, A. Jacob, T. Chen, B. Rosenburg, O. Sallénave, C. Bertolli, S. Antao, J. Brunheroto, Y. Park, K. O'Brien, and R. Nair, "Data Access Optimization in a Processing-in-memory System," in *CF*, 2015.
 - [145] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM JRD*, 1967.
 - [146] Transaction Processing Performance Council, "http://www.tpc.org/.
 - [147] A. N. Udupi, N. Muralimanohar, N. Chatterjee, R. Balasubramanian, A. Davis, and N. P. Jouppi, "Rethinking DRAM Design and Organization for Energy-constrained Multi-cores," in *ISCA*, 2010.
 - [148] H. Usui, L. Subramanian, K. K. Chang, and O. Mutlu, "SQUASH: Simple QoS-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators," *arXiv:1505.07502 [cs.AR]*, 2015.
 - [149] H. Usui, L. Subramanian, K. K. Chang, and O. Mutlu, "DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators," *TACO*, 2016.
 - [150] T. Vogelsang, "Understanding the Energy Consumption of Dynamic Random Access Memories," in *MICRO*, 2010.
 - [151] F. A. Ware and C. Hampel, "Improving Power and Data Efficiency with Threaded Memory Modules," in *ICCD*, 2006.
 - [152] W. A. Wong and J.-L. Baer, "DRAM caching," 1997.
 - [153] D. Xiong, K. Huang, X. Jiang, and X. Yan, "Memory Access Scheduling Based on Dynamic Multilevel Priority in Shared DRAM Systems," *TACO*, 2016.
 - [154] T. Yamauchi, L. Hammond, and K. Olukotun, "The Hierarchical Multi-bank DRAM: A High-performance Architecture for Memory Integrated with Processors," in *Advanced Research in VLSI*, 1997.
 - [155] H. Yoon, J. Meza, R. Ausavarungnirun, R. Harding, and O. Mutlu, "Row Buffer Locality Aware Caching Policies for Hybrid Memories," in *ICCD*, 2012.
 - [156] J. Yue and Y. Zhu, "Exploiting Subarrays Inside a Bank to Improve Phase Change Memory Performance," in *DATE*, 2013.
 - [157] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, "TOP-PIM: Throughput-oriented Programmable Processing in Memory," in *HPDC*, 2014.
 - [158] L. Zhang, Z. Fang, M. Parker, B. K. Mathew, L. Schaelicke, J. B. Carter, W. C. Hsieh, and S. A. McKee, "The Impulse Memory Controller," *IEEE TC*, 2001.
 - [159] T. Zhang, M. Poremba, C. Xu, G. Sun, and Y. Xie, "CREAM: A Concurrent-Refresh-Aware DRAM Memory Architecture," in *HPCA*, 2014.
 - [160] Z. Zhang, Z. Zhu, and X. Zhang, "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality," in *MICRO*, 2000.
 - [161] Z. Zhang, Z. Zhu, and X. Zhang, "Cached DRAM for ILP processor memory access latency reduction," *IEEE Micro*, July 2001.
 - [162] J. Zhao, O. Mutlu, and Y. Xie, "FIRM: Fair and High-Performance Memory Control for Persistent Memory Systems," in *MICRO*, 2014.
 - [163] L. Zhao, R. Iyer, S. Makineni, L. Bhuyan, and D. Newell, "Hardware Support for Bulk Data Movement in Server Platforms," in *ICCD*, 2005.
 - [164] H. Zheng, J. Lin, Z. Zhang, E. Gorbato, H. David, and Z. Zhu, "Mini-rank: Adaptive DRAM Architecture for Improving Memory Power Efficiency," in *MICRO*, 2008.

Reducing DRAM Refresh Overheads with Refresh-Access Parallelism

Kevin K. Chang^{1,2} Donghyuk Lee^{3,2} Zeshan Chishti⁴
Alaa R. Alameldeen⁴ Chris Wilkerson⁴ Yoongu Kim² Onur Mutlu^{5,2}

¹Facebook ²Carnegie Mellon University ³NVIDIA Research ⁴Intel Labs ⁵ETH Zürich

This article summarizes the idea of “refresh-access parallelism,” which was published in HPCA 2014 [17], and examines the work’s significance and future potential. The overarching objective of our HPCA 2014 paper is to reduce the significant negative performance impact of DRAM refresh with intelligent memory controller mechanisms.

To mitigate the negative performance impact of DRAM refresh, our HPCA 2014 paper proposes two complementary mechanisms, DARP (Dynamic Access Refresh Parallelization) and SARP (Subarray Access Refresh Parallelization). The goal is to address the drawbacks of state-of-the-art per-bank refresh mechanism by building more efficient techniques to parallelize refreshes and accesses within DRAM. First, instead of issuing per-bank refreshes in a round-robin order, as it is done today, DARP issues per-bank refreshes to idle banks in an out-of-order manner. Furthermore, DARP proactively schedules refreshes during intervals when a batch of writes are draining to DRAM. Second, SARP exploits the existence of mostly-independent subarrays within a bank. With minor modifications to DRAM organization, it allows a bank to serve memory accesses to an idle subarray while another subarray is being refreshed. Our extensive evaluations on a wide variety of workloads and systems show that our mechanisms improve system performance (and energy efficiency) compared to three state-of-the-art refresh policies, and their performance benefits increase as DRAM density increases.

1. Introduction

Modern main memory is predominantly built using dynamic random access memory (DRAM) cells. A DRAM cell consists of a capacitor to store one bit of data as electrical charge. The capacitor leaks charge over time, causing stored data to change. As a result, DRAM requires an operation called *refresh* that periodically restores electrical charge in DRAM cells to maintain data integrity.

There are two major ways refresh operations are performed in modern DRAM systems: *all-bank refresh* (or, *rank-level refresh*) and *per-bank refresh*. These methods differ in what levels of the DRAM hierarchy refresh operations tie up. A modern DRAM system is organized as a hierarchy of ranks and banks. Each rank is composed of multiple banks. Different ranks and banks can be accessed independently. Each bank contains a number of rows (e.g., 16-32K in modern chips). Because successively refreshing *all* rows in a DRAM chip would cause very high delay by tying up the entire DRAM

device, modern memory controllers issue a number of refresh commands that are evenly distributed throughout the refresh interval [38, 40, 73, 74, 93]. Each refresh command refreshes a small number of rows.¹ The two common refresh methods of today differ in where in the DRAM hierarchy the rows refreshed by a refresh command reside.

In *all-bank refresh* (REF_{ab}), employed by both commodity DDR and LPDDR DRAM chips, a refresh command operates at the rank level: it refreshes a number of rows in *all* banks of a rank concurrently. This causes every bank within a rank to be unavailable to serve memory requests until the refresh command is complete. Therefore, it degrades performance significantly [4, 17, 74, 88, 93, 96, 115].

An alternative method is to perform refresh operations at the bank level, called *per-bank refresh* (REF_{pb}), which is currently supported in LPDDR DRAM used in mobile platforms [40]. In contrast to REF_{ab} , REF_{pb} enables a bank to be accessed while another bank is being refreshed, alleviating part of the negative performance impact of refresh. Figure 1 shows pictorially how REF_{pb} provides performance benefits over REF_{ab} from parallelization of refreshes and reads. REF_{pb} reduces refresh interference on reads by issuing a refresh to Bank 0 while Bank 1 is serving reads. Subsequently, it refreshes Bank 1 while allowing Bank 0 to serve a read. As a result, REF_{pb} alleviates part of the performance loss due to refreshes by enabling parallelization of refreshes and accesses across banks.

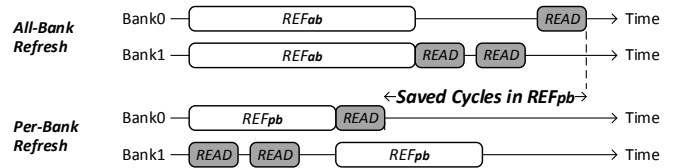


Figure 1: Service timelines of all-bank and per-bank refresh. Adapted from [17].

Unfortunately, there are two shortcomings of per-bank refresh. First, refreshes to different banks are scheduled in a strict round-robin order, as specified by the LPDDR standard [40]. Using this static policy may force a busy bank to be refreshed, delaying the memory requests queued in that

¹The time between two refresh commands is fixed to an amount that is dependent on the DRAM type and temperature. We refer the reader to our prior works [17, 18, 19, 20, 30, 31, 49, 52, 53, 54, 55, 56, 67, 68, 69, 70, 71, 73, 74, 96, 107, 108] for a detailed background on DRAM.

bank, while other idle banks are available to be refreshed. Second, a bank that is refreshing *cannot* concurrently serve memory requests. Hence, requests to a refreshing bank get delayed due to a “refresh-access bank conflict.”

We show that the negative performance impact of DRAM refresh becomes exacerbated as DRAM density increases in the future. Figure 2 shows the average performance degradation of all-bank/per-bank refresh compared to ideal baseline without any refresh.² Although REF_{pb} performs slightly better than REF_{ab} , the performance loss due to refresh is still significant, especially as the density grows (16.6% loss at 32Gb). Therefore, **the goal** this work is to provide practical mechanisms to overcome the aforementioned two shortcomings to mitigate the performance overhead of DRAM refresh.

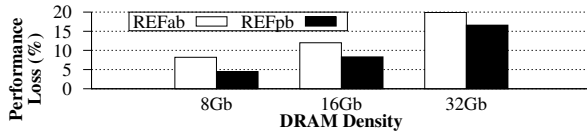


Figure 2: Performance loss due to all-bank refresh (REF_{ab}) and per-bank refresh (REF_{pb}). Reproduced from [17].

2. Parallelizing Refreshes with Memory Accesses

We propose two mechanisms, *Dynamic Access Refresh Parallelization (DARP)* and *Subarray Access Refresh Parallelization (SARP)*, that hide refresh latency by parallelizing refreshes with memory accesses across *banks* and *subarrays*, respectively. In this section, we present a brief overview of these two new mechanisms. We refer the reader to Section 4 of our HPCA 2014 paper [17] for more detail on the algorithm and implementation.

2.1. Dynamic Access Refresh Parallelization (DARP)

DARP is a new refresh scheduling policy that consists of two components. The first component is *out-of-order per-bank refresh*, which enables the memory controller to specify a particular (idle) bank to be refreshed as opposed to the standard per-bank refresh policy that refreshes banks in a strict round-robin order. With out-of-order refresh scheduling, DARP can avoid refreshing (non-idle) banks with pending memory requests, thereby avoiding the refresh latency for those requests. The second component is *write-refresh parallelization* that proactively issues REF_{pb} to a bank while DRAM is draining write batches to other banks, thereby overlapping refresh latency with write request latencies.

2.1.1. DARP: Out-of-order Per-bank Refresh. A major limitation of the current REF_{pb} mechanism is that it disallows a memory controller from specifying which bank to refresh. Instead, a DRAM chip has internal logic that strictly refreshes banks in a *sequential round-robin order*. Because DRAM

lacks visibility into a memory controller’s state (e.g., request queues’ occupancy), simply using an in-order REF_{pb} policy can unnecessarily refresh a bank that has multiple pending requests to be served when other banks may be *free* to serve a refresh command. To address this problem, we propose the first component of DARP, *out-of-order per-bank refresh*. The idea is to remove the bank selection logic from DRAM and make it the memory controller’s responsibility to determine which bank to refresh. As a result, the memory controller can refresh an idle bank to enhance parallelization of refreshes and accesses, avoiding refreshing a bank that has pending requests as much as possible.

Due to REF_{pb} reordering, the memory controller needs to guarantee that deviating from the original in-order refresh schedule still preserves data integrity. To achieve this, we take advantage of the fact that the contemporary DDR JEDEC standard [39] provides some refresh scheduling flexibility. The standard allows up to *eight* all-bank refresh commands to be issued late (postponed) or early (pulled-in). This implies that each bank can tolerate up to eight REF_{pb} commands to be postponed or pulled in. Therefore, the memory controller ensures that reordering REF_{pb} preserves data integrity by limiting the number of postponed or pulled-in commands. Our HPCA 2014 paper [17] describes our new algorithm for out-of-order per-bank refresh in detail.

2.1.2. DARP: Write-refresh Parallelization. The key idea of the second component of DARP is to actively avoid refresh interference on read requests and instead enable more parallelization of refreshes with *write requests*. We make two observations that lead to our idea. First, *write batching* in DRAM [65] creates an opportunity to overlap a refresh operation with a sequence of writes, without interfering with reads. A modern memory controller typically buffers DRAM writes and drains them to DRAM in a batch to amortize the *bus turnaround latency*, also called *tWTR* or *tRTW* [39, 56, 65], which is the additional latency incurred from switching between serving writes to reads and vice versa. Typical systems start draining writes when the write buffer occupancy exceeds a certain threshold until the buffer reaches a low watermark. This draining time period is called the *writeback mode*, during which no rank within the draining channel can serve read requests [22, 65, 116]. Second, DRAM writes are usually *not* latency-critical because processors do not stall to wait for them: DRAM writes are due to dirty cache line evictions from the last-level cache [65, 105, 116].

Given that writes are not latency-critical and are drained in a batch for some time interval, they are more flexible to be scheduled with minimal performance impact. We propose the second component of DARP, *write-refresh parallelization*, that attempts to maximize parallelization of refreshes and writes. Write-refresh parallelization selects the bank with the minimum number of pending demand requests (both read and write) and preempts the bank’s writes with a per-bank

²Our detailed methodology is described in Section 5 of our full paper [17].

refresh. As a result, the bank's refresh operation is hidden by the writes in other banks.

Figure 3 shows the service timeline and benefits of write-refresh parallelization. There are **two scenarios** when the scheduling policy parallelizes refreshes with writes to increase DRAM's availability to serve read requests. Figure 3a shows the first scenario when the scheduler *postpones* issuing a REF_{pb} command to avoid delaying a read request in Bank 0 and instead serves the refresh in parallel with writes from Bank 1, effectively hiding the refresh latency in the writeback mode. Even though the refresh can potentially delay individual write requests during writeback mode, the delay does not impact performance as long as the length of writeback mode remains the same as in the baseline due to longer prioritized write request streams in other banks. In the second scenario shown in Figure 3b, the scheduler *pulls in* a REF_{pb} command early in Bank 0 to fully hide the refresh latency from the later read request while Bank 1 is draining writes during the writeback mode (note that the read request cannot be scheduled during the writeback mode).

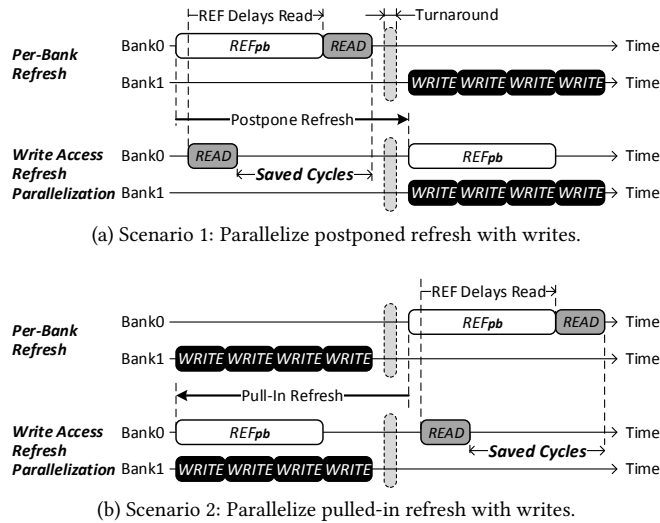


Figure 3: Service timeline of a per-bank refresh operation along with read and write requests using different refresh scheduling policies. Reproduced from [17].

2.2. Subarray Access Refresh Parallelization (SARP)

To tackle the problem of refreshes and accesses colliding within the same bank, we propose *SARP (Subarray Access Refresh Parallelization)*, which exploits the existence of subarrays [56] within a bank. A DRAM bank is sub-divided into multiple *subarrays* [19, 23, 31, 56, 67, 69, 70, 76, 106, 107, 108, 110, 120, 125, 126], as shown in Figure 4. A subarray consists of a 2-D array of cells organized in rows and columns.³ Each DRAM cell has two components: 1) a *capacitor* that stores one

³Physically, DRAM has 32 to 128 subarrays, which varies depending on the number of rows (typically 16-64K) within a bank. This work divides them into 8 *subarray groups*. We refer to a subarray group as a subarray [56], without loss of generality.

bit of data as electrical charge, and 2) an *access transistor* that connects the capacitor to a wire called *bitline* that is shared by a column of cells. The access transistor is controlled by a wire called *wordline* that is shared by a row of cells. When a wordline is raised to V_{DD} , a row of cells becomes connected to the bitlines, allowing reading or writing data to the connected row of cells. The component that reads (i.e., senses) or writes a bit of data on a bitline is called a *sense amplifier*, shared by an entire column of cells. A row of sense amplifiers is also called a *row buffer*. All subarrays' row buffers are connected to an I/O buffer [22, 48, 68, 87] that reads and writes data from/to the bank's I/O bus.

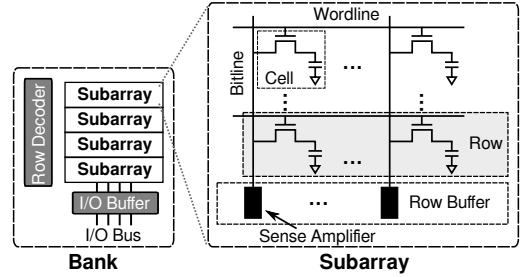


Figure 4: DRAM bank and subarray organization. Reproduced from [17].

The key observation leading to our second mechanism, SARP, is that a refresh operation is constrained to only a few *subarrays* within a bank whereas the other *subarrays* and the *I/O bus* remain idle during the process of refreshing. The reasons for this are two-fold. First, refreshing a row requires only its subarray's sense amplifiers that restore the charge in the row *without* transferring any data through the I/O bus. Second, each subarray has its own set of *sense amplifiers* that are *not* shared with other subarrays.

Based on this observation, SARP's key idea is to allow memory accesses to an *idle* subarray while other subarrays are refreshing. Figure 5 shows the service timeline and the performance benefit of our mechanism. As shown, SARP reduces the read latency by performing the read operation to Subarray 1 in parallel with the refresh in Subarray 0. Compared to DARP, SARP provides the following advantages: 1) SARP is applicable to both all-bank and per-bank refresh, 2) SARP enables memory accesses to a refreshing bank, which cannot be achieved with DARP, and 3) SARP also utilizes bank-level parallelism [66, 91] by serving memory requests to multiple banks in parallel while the entire rank is under refresh.

SARP requires modifications to 1) the DRAM architecture, because two distinct wordlines in different subarrays need to be raised simultaneously (to accommodate parallel refresh and access to the two subarrays), which cannot be done in today's DRAM due to the shared peripheral logic among subarrays; and 2) the memory controller, such that it can keep track of which subarray is under refresh in order to send the appropriate memory request to an idle subarray. Section 4.3 of our HPCA 2014 paper [17] describes these changes in detail. To evaluate the benefits and die area overhead of SARP,

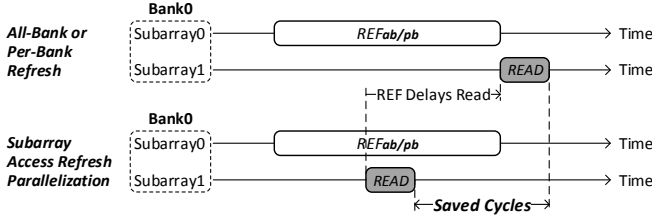


Figure 5: Service timeline of a refresh and a read request to two different subarrays within the same bank. Reproduced from [17].

we use 8 subarrays per bank and 8 banks per DRAM chip. Based on this configuration, we calculate the area overhead of SARP using parameters from a Rambus DRAM model at 55nm technology [101], and find it to be 0.71% in a 2Gb DDR3 DRAM chip with a die area of 73.5mm². The power overhead of the additional components is negligible compared to the entire DRAM chip.

3. Evaluation

We briefly summarize our results on an eight-core system. Section 6 of our HPCA 2014 paper provides detailed evaluations on a wide variety of systems and sensitivity studies. We evaluate the performance of our proposed mechanisms on an eight-core system using Ramulator [52, 103], an open-source cycle-level DRAM simulator, driven by CPU traces generated from Pin [77]. We use benchmarks from SPEC CPU2006 [113], STREAM [83], TPC [118], and a microbenchmark with random-access behavior similar to HPCC RandomAccess [34]. Table 1 summarizes the configuration of our evaluated system.

Processor	8 cores, 4GHz, 3-wide issue, 8 MSHRs/core, 128-entry instruction window
Last-level Cache	64B cache-line, 16-way associative, 512KB private cache-slice per core
Memory Controller	64/64-entry read/write request queue, FR-FCFS [102], writes are scheduled in batches [22, 65, 116] with low watermark = 32, closed-row policy [22, 54, 55, 102]
DRAM	DDR3-1333 [86], 2 channels, 2 ranks per channel, 8 banks/rank, 8 subarrays/bank, 64K rows/bank, 8KB rows
Refresh Settings	$tRFC_{ab} = 350/530/890$ ns for 8/16/32Gb DRAM chips, $tREFI_{ab} = 3.9\mu$ s, $tRFC_{ab}$ -to- $tRFC_{pb}$ ratio = 2.3

Table 1: Evaluated system configuration. Adapted from [17].

Figure 6 shows the average system performance (left) and energy per DRAM access (right) of our final mechanism, DSARP, the combination of DARP and SARP, compared to two baseline refresh schemes and an ideal scheme without any refreshes. We measure system performance with the commonly-used *weighted speedup* (WS) [26, 109] metric. The percentage numbers on top of the bars are the performance improvement of DSARP over REF_{ab} .

We make two observations. First, DSARP consistently improves system performance and energy efficiency over prior refresh schemes, capturing most of the benefit of the ideal system with no refresh. Second, as DRAM density (i.e., refresh

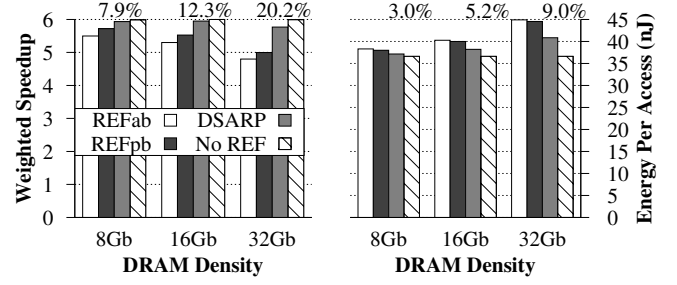


Figure 6: Average system performance and energy consumption due to different refresh mechanisms.

latency) increases, the performance benefit of DSARP gets larger. We conclude that DSARP is an effective mechanism to alleviate the negative performance impact of DRAM refresh.

3.1. Comparison to DDR4 Fine Granularity Refresh

DDR4 DRAM supports a new refresh mode called *fine granularity refresh* (FGR) in an attempt to mitigate the increasing refresh latency ($tRFC_{ab}$) [39]. FGR trades off shorter $tRFC_{ab}$ with a faster refresh rate ($1/tREFI_{ab}$) that increases by either 2x or 4x. Figure 7 shows the effect of FGR in comparison to REF_{ab} , *adaptive refresh policy* (AR) [88], and DSARP. 2x and 4x FGR actually *reduce* average system performance by 3.9%/4.0%/4.3% and 8.1%/13.7%/15.1% compared to REF_{ab} with 8/16/32Gb densities, respectively. As the refresh rate increases by 2x/4x (higher refresh penalty), $tRFC_{ab}$ does *not* scale down with the same constant factors. Instead, $tRFC_{ab}$ reduces by 1.35x/1.63x with 2x/4x higher rate [39], thus increasing the worst-case refresh latency by 1.48x/2.45x. This performance degradation due to FGR has also been observed in Mukundan et al. [88]. AR [88] dynamically switches between 1x (i.e., REF_{ab}) and 4x refresh modes to mitigate the downsides of FGR. AR performs slightly worse than REF_{ab} (within 1%) for all densities. Because using 4x FGR greatly degrades performance, AR can only mitigate the large loss from the 4x mode and cannot improve performance over REF_{ab} . On the other hand, DSARP is a more effective mechanism to tolerate the long refresh latency than both FGR and AR as it overlaps refresh latency with access latency without increasing the refresh rate.

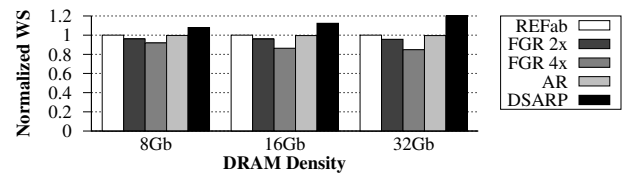


Figure 7: Performance comparisons to FGR and AR [88]. Reproduced from [17].

We conclude that DSARP is an effective mechanism that can effectively tolerate and hide longer refresh latencies, which are expected in future DRAM devices as DRAM technology scales to even smaller feature sizes.

4. Related Work

To our knowledge, this is the first work to comprehensively study the effect of per-bank refresh and propose 1) a refresh scheduling policy built on top of per-bank refresh and 2) a mechanism that achieves parallelization of refresh operations and memory accesses *within* a refreshing bank. We discuss prior works that mitigate the negative effects of DRAM refresh and compare them to our mechanisms.

Retention-Aware Refresh. Various works (e.g., [1, 3, 4, 5, 27, 50, 72, 74, 94, 95, 96, 98, 119]) propose mechanisms to reduce unnecessary refresh operations by taking advantage of the fact that different DRAM cells have widely different retention times [51, 73, 96]. These works assume that the retention time of DRAM cells can be *accurately* profiled and they depend on having this accurate profile to guarantee data integrity [73]. However, as shown in Liu et al. [73] and later analyzed in detail by several other works [44, 45, 46, 47, 96, 98], accurately determining the retention time profile of DRAM is an outstanding research problem due to the Variable Retention Time (VRT) and Data Pattern Dependence (DPD) phenomena, which can cause the retention time of a cell to fluctuate over time. As such, retention-aware refresh techniques need to overcome the profiling challenges to be viable. A recent work, AVATAR [98], proposes a retention-aware refresh mechanism that addresses VRT by using ECC chips, which introduces extra cost. In contrast, our refresh mitigation techniques enable parallelization of refreshes and accesses *without* relying on cell data retention profiles or ECC, thus providing high reliability at low cost.

Refresh Scheduling. Stuecheli et al. [115] propose elastic refresh that postpones refreshes by a time delay that varies based on the number of postponed refreshes and the predicted rank idle time to avoid interfering with demand requests. Elastic refresh has two shortcomings. First, it becomes less effective when the average rank idle period is shorter than $tRFC_{ab}$ as the refresh latency cannot be fully hidden in that period. This occurs especially with 1) more memory-intensive workloads that inherently have less idleness and 2) higher density DRAM chips that have higher $tRFC_{ab}$. Second, elastic refresh incurs more refresh latency when it *incorrectly* predicts a time period as idle when the time period actually has pending requests. In contrast, our mechanisms parallelize refresh operations with accesses even if there is no idle period and therefore outperform elastic refresh.

Ishii et al. [37] propose a write scheduling policy that prioritizes write draining over read requests in a rank while another rank is refreshing (even if the write queue has not reached the threshold to trigger write mode). This technique is *only* applicable in multi-ranked memory systems. Our mechanisms are *also* applicable to single-ranked memory systems by enabling parallelization of refreshes and accesses at the bank and subarray levels, and they can be combined with Ishii et al. [37].

Mukundan et al. [88] propose scheduling techniques (in addition to adaptive refresh discussed in Section 3.1) to address the problem of *command queue seizure*, whereby a command queue gets filled up with commands to a refreshing rank, blocking commands to *another* non-refreshing rank. In our work, we use a different memory controller design that does not have command queues, similarly to prior work [32]. Our controller generates a command for a scheduled request *right before* the request is sent to DRAM instead of pre-generating the commands and queuing them up. Thus, our baseline design does not suffer from the problem of command queue seizure.

Subarray-Level Parallelism (SALP). Kim et al. [56] propose SALP to reduce bank serialization latency by enabling *multiple accesses* to different subarrays within a bank to proceed in a pipelined manner. In contrast to SALP, our mechanism (SARP) parallelizes *refreshes and accesses* to different subarrays within the same bank. Therefore, SARP exploits the existence of subarrays for a different purpose and in a different way from SALP. We reduce the sharing of the peripheral circuits for refreshes and accesses, not for arbitrary accesses. As such, our implementation is not only different, but also less intrusive than SALP: SARP does not require new DRAM commands and timing constraints. We note that several other works exploit the existence of subarrays for various performance and energy improvement purposes [19, 67, 69, 70, 106, 107, 108]. We refer the reader to the SALP paper in this very same issue for a detailed treatment of SALP [57].

DRAM Refresh Architecture. Several other works propose different refresh architectures. Nair et al. [93] propose Refresh Pausing, which pauses a refresh operation to serve pending memory requests when the refresh causes conflicts with the requests. Although our work already significantly reduces conflicts between refreshes and memory requests by enabling parallelization, it can be combined with Refresh Pausing to address rare conflicts. Tavva et al. [117] propose EFGR, which exposes non-refreshing banks during an all-bank refresh operation so that a few accesses can be scheduled to those non-refresh banks during the refresh operation. However, such a mechanism does not provide additional performance and energy benefits over per-bank refresh, which we use to build our mechanism in this dissertation. Isen and John [36] propose ESKIMO, which modifies the ISA to enable memory allocation libraries to skip refreshes on memory regions that do not affect programs' execution. ESKIMO is orthogonal to our mechanism, and its modification has high system-level complexity by requiring system software libraries to make refresh decisions. Other techniques (e.g., heterogeneous-reliability memory [81] or Flicker [75]) can eliminate or reduce refreshes in parts of memory. Our techniques are complementary to such refresh elimination/reduction techniques.

eDRAM Concurrent Refresh. Kiriata et al. [58] propose a mechanism to enable a bank to refresh independently while another bank is being accessed in embedded DRAM (eDRAM). Our work differs from [58] in two major ways. First, unlike SARP, [58] parallelizes refreshes only across banks, not *within* each bank. Second, there are significant differences between DRAM and eDRAM architectures, which make it non-trivial to apply [58]’s mechanism directly to DRAM. In particular, eDRAMs have no standardized timing/power integrity constraints and access protocol, making it simpler for each bank to independently manage its refresh schedule. In contrast, refreshes in DRAM need to be managed by the memory controller to ensure that parallelizing refreshes with accesses does not violate other constraints. Other works (e.g., [2, 25]) exploit the fact that eDRAM is used as a cache to avoid refresh operations.

5. Significance

In this section, we describe three trends in the current and future DRAM subsystem that will likely make our proposed solutions more important and attractive in the future, and examine the work’s impact on future research.

5.1. Long-Term Impact

Worsening Retention Time. As the DRAM cell feature size continues to scale, the cells’ retention time will likely become shorter, exacerbating the refresh penalty [43, 89, 90]. When the surface area of cells gets smaller with further scaling, the depth/height of the cell needs to increase to maintain the same amount of capacitance that can be stored in a cell. In other words, the *aspect ratio* (the ratio of a cell’s depth to its diameter) needs to be increased to maintain the capacitance. However, many works have shown that fabricating high aspect ratio cells is becoming more difficult due to processing technology [33, 43, 82]. Therefore, the cells’ capacitance (and, thus, their retention time) may potentially decrease with further scaling, increasing the refresh frequency. Using DSARP is a cost-effective way to alleviate the increasing negative impact of refresh as our results show [17]. Note that errors have started appearing in DRAM chips due to aggressive technology scaling [53, 85, 89, 104, 111, 112]. The RowHammer problem is a prime example of DRAM errors that have been slipping into the field [53, 89], and one solution for it is to increase the refresh rate [53, 89]. Such solutions to technology scaling issues clearly exacerbate the refresh problem. Therefore, DSARP can alleviate the performance impact under these conditions.

New DRAM Standards with Flexible Per-Bank Refresh. According to newer DRAM standards, the industry is already in the process of implementing a similar concept of enabling the memory controller to determine which bank to refresh. In particular, the two standards are: 1) HBM [41, 71] (October 2013, after the submission of our HPCA 2014 paper [17]) and 2) LPDDR4 [42] (August 2014). Both standards have incorporated a new refresh mode that allows

per-bank refresh commands to be issued in *any* order by the memory controllers. Neither standard specifies a preferred order which the memory controller needs to follow for issuing refresh commands.

Our work has done extensive evaluations to show that our proposed per-bank refresh scheduling policy, *DARP*, outperforms a naive round-robin policy by opportunistically refreshing idle banks. As a result, our policy can be potentially adopted in the future processors that use HBM or LPDDR4 DRAM.

Increasing Number of Subarrays. As DRAM density keeps increasing, more rows of cells are added within each DRAM bank. To avoid the disadvantage of increasing sensing latency due to longer bitlines in subarrays [18, 70], more subarrays will likely be added within a single bank instead of increasing the size of each subarray. Our proposed refreshing scheme at the subarray level, *SARP*, becomes more effective at mitigating refresh as the number of subarrays increases because the probability of a refresh and a demand request colliding at the subarray level decreases with more subarrays.

5.2. Potential Research Impact

Impact on Recent Research Work. To our knowledge, this is the first work to comprehensively study and extend the concept of *per-bank refresh* to DDRx DRAM chips. Several works [5, 28, 117] use our per-bank refresh mechanism as a baseline for comparison. Kotra et al. [60] propose a new refresh mechanism to further enhance our per-bank refresh mechanism. Kong et al. [59] extend our per-bank refresh idea to eDRAM.

Future Research Directions. This work will likely create new research opportunities for studying refresh scheduling policies at different dimensions (i.e., bank and subarray level) to mitigate worsening refresh overheads. Among many potential opportunities, one potential way to further reduce the refresh latency (i.e., $tRFC_{ab/pt}$) is to trade off higher refresh rate (i.e., $tREFI$), which is currently supported as *fine granularity refresh* in DDR4 DRAM for all-bank refresh. In this work, we assume a fixed refresh rate for per-bank refresh as it is specified in the standard. Therefore, a new research question that our work raises is *how can one combine per-bank refresh with fine granularity refresh and design a new scheduling policy for that?* We think that *DARP* can inspire new scheduling policies to improve the performance of existing DRAM designs.

Applicability to Other Memory Technologies. Refresh is used in NAND flash memory to improve lifetime [12, 13, 14, 78], and can be used as a general solution to several other NAND flash reliability problems that are characterized and discussed in various recent works [6, 7, 8, 9, 10, 11, 15, 16, 79, 80]. We believe the idea of DSARP and refresh scheduling can also be applied to refresh mechanisms in flash memory, and this can be especially beneficial toward the end of the lifetime of flash memory when the device is refreshed more

frequently [7, 8, 9, 13]. We refer the reader to our recent works to understand the mechanisms for refresh in modern flash memories [7, 8, 9].

We believe the principles of DSARP are also applicable to emerging memory technologies [84], e.g., phase-change memory (PCM) [62, 63, 64, 99, 100, 122, 123, 124], STT-MRAM [21, 29, 61, 92], or RRAM/memristors [24, 114, 121]. For example, PCM suffers from *resistance drift* [35, 97, 122], where the resistance used to represent the value becomes higher over time (and eventually can introduce a bit error). To mitigate resistance drift, PCM can use refresh-like operations to rewrite the original data value, and as the density of PCM grows, more such operations are required. We leave a detailed exploration of how DSARP can be used for emerging memory technologies to future works.

6. Conclusion

We introduced two new complementary techniques, DARP (Dynamic Access Refresh Parallelization) and SARP (Subarray Access Refresh Parallelization), to mitigate the DRAM refresh penalty by enhancing *refresh-access parallelization* at the bank and subarray levels, respectively. DARP 1) issues per-bank refreshes to idle banks in an out-of-order manner instead of issuing refreshes in a strict round-robin order, 2) proactively schedules per-bank refreshes during intervals when a batch of writes are draining to DRAM. SARP enables a bank to serve requests from idle subarrays in parallel with other subarrays that are being refreshed. Our extensive evaluations on a wide variety of systems and workloads show that these mechanisms significantly improve system performance and outperform state-of-the-art refresh policies, approaching the performance of ideally eliminating all refreshes. We conclude that DARP and SARP are effective at hiding the refresh latency penalty in modern and near-future DRAM systems, and that their benefits increase as DRAM density increases.

We believe these techniques are also applicable to other memory technologies, such as NAND flash memory and phase change memory. We hope our work inspires future research to develop even more effective refresh latency tolerance techniques.

Acknowledgments

We thank Saugata Ghose for his dedicated effort in the preparation of this article. We thank the anonymous reviewers and Jamie Liu for helpful feedback and the members of the SAFARI research group for feedback and the stimulating environment they provide. We acknowledge the support of IBM, Intel, and Samsung. This research was supported in part by the Intel Science and Technology Center on Cloud Computing, the Semiconductor Research Corporation, and an NSF CAREER Award (grant 0953246).

References

- [1] A. Agrawal *et al.*, "Mosaic: Exploiting the spatial locality of process variation to reduce refresh energy in on-chip eDRAM modules," in *HPCA*, 2014.

- [2] A. Agrawal *et al.*, "Refrint: Intelligent Refresh to Minimize Power in On-Chip Multiprocessor Cache Hierarchies," in *HPCA*, 2013.
- [3] A. Agrawal *et al.*, "CLARA: Circular Linked-List Auto and Self Refresh Architecture," in *MEMSYS*, 2016.
- [4] S. Baek *et al.*, "Refresh Now and Then," *IEEE TC*, vol. 63, no. 12, pp. 3114–3126, 2014.
- [5] I. Bhati *et al.*, "Flexible Auto-refresh: Enabling Scalable and Energy-efficient DRAM Refresh Reductions," in *ISCA*, 2015.
- [6] Y. Cai *et al.*, "Read Disturb Errors in MLC NAND Flash Memory: Characterization and Mitigation," in *DSN*, 2015.
- [7] Y. Cai *et al.*, "Error Characterization, Mitigation, and Recovery in Flash-Memory-Based Solid-State Drives," *Proceedings of the IEEE*, 2017.
- [8] Y. Cai *et al.*, "Error Characterization, Mitigation, and Recovery in Flash Memory Based Solid-State Drives," arXiv:1706.08642 [cs.AR], 2017.
- [9] Y. Cai *et al.*, "Errors in Flash-Memory-Based Solid-State Drives: Analysis, Mitigation, and Recovery," arXiv:1711.11427 [cs.AR], 2017.
- [10] Y. Cai *et al.*, "Vulnerabilities in MLC NAND Flash Memory Programming: Experimental Analysis, Exploits, and Mitigation Techniques," in *HPCA*, 2017.
- [11] Y. Cai *et al.*, "Error Patterns in MLC NAND Flash Memory: Measurement, Characterization, and Analysis," in *DATE*, 2012.
- [12] Y. Cai *et al.*, "Data Retention in MLC NAND Flash Memory: Characterization, Optimization, and Recovery," in *HPCA*, 2015.
- [13] Y. Cai *et al.*, "Flash Correct-and-Refresh: Retention-Aware Error Management for Increased Flash Memory Lifetime," in *JCCD*, 2012.
- [14] Y. Cai *et al.*, "Error Analysis and Retention-Aware Error Management for NAND Flash Memory," in *ITJ*, 2013.
- [15] Y. Cai *et al.*, "Neighbor Cell Assisted Error Correction in MLC NAND Flash Memories," in *SIGMETRICS*, 2014.
- [16] Y. Cai *et al.*, "Threshold Voltage Distribution in MLC NAND Flash Memory: Characterization, Analysis, and Modeling," in *DATE*, 2013.
- [17] K. K. Chang *et al.*, "Improving DRAM Performance by Parallelizing Refreshes with Accesses," in *HPCA*, 2014.
- [18] K. K. Chang *et al.*, "Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization," in *SIGMETRICS*, 2016.
- [19] K. K. Chang *et al.*, "Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM," in *HPCA*, 2016.
- [20] K. K. Chang *et al.*, "Understanding Reduced-Voltage Operation in Modern DRAM Devices: Experimental Characterization, Analysis, and Mechanisms," in *SIGMETRICS*, 2017.
- [21] M. T. Chang *et al.*, "Technology Comparison for Large Last-Level Caches (L3Cs): Low-Leakage SRAM, Low Write-Energy STT-RAM, and Refresh-Optimized eDRAM," in *HPCA*, 2013.
- [22] N. Chatterjee *et al.*, "Staged reads: Mitigating the impact of DRAM writes on DRAM reads," in *HPCA*, 2012.
- [23] J. Choi *et al.*, "Multiple Clone Row DRAM: A Low Latency and Area Optimized DRAM," in *ISCA*, 2015.
- [24] L. Chua, "Memristor – The Missing Circuit Element," *TCT*, 1971.
- [25] P. G. Emma *et al.*, "Rethinking Refresh: Increasing Availability and Reducing Power in DRAM for Cache Applications," *IEEE Micro*, vol. 28, no. 6, pp. 47–56, 2008.
- [26] S. Eyerman and L. Eeckhout, "System-Level Performance Metrics for Multiprogram Workloads," *IEEE Micro*, 2008.
- [27] Y. H. Gong and S. W. Chung, "Exploiting Refresh Effect of DRAM Read Operations: A Practical Approach to Low-Power Refresh," *IEEE TC*, vol. 65, no. 5, pp. 1507–1517, 2016.
- [28] M. Guan and L. Wang, "Temperature aware refresh for DRAM performance improvement in 3D ICs," in *ISQED*, 2015.
- [29] X. Guo *et al.*, "Resistive Computation: Avoiding the Power Wall with Low-Leakage, STT-MRAM Based Computing," in *ISCA*, 2010.
- [30] H. Hassan *et al.*, "SoftMC: A Flexible and Practical Open-Source Infrastructure for Enabling Experimental DRAM Studies," in *HPCA*, 2017.
- [31] H. Hassan *et al.*, "ChargeCache: Reducing DRAM Latency by Exploiting Row Access Locality," in *HPCA*, 2016.
- [32] E. Herrero *et al.*, "Thread row buffers: Improving memory performance isolation and throughput in multiprogrammed environments," *IEEE TC*, vol. 62, no. 9, pp. 1879–1892, 2013.
- [33] S. Hong, "Memory technology trend and future challenges," in *IEDM*, 2010.
- [34] HPC Challenge, "RandomAccess," <http://icl.cs.utk.edu/hpcc>.
- [35] D. Ielmini *et al.*, "Recovery and Drift Dynamics of Resistance and Threshold Voltages in Phase-Change Memories," *TED*, 2007.
- [36] C. Isen and L. John, "ESKIMO - energy savings using semantic knowledge of inconsequential memory occupancy for DRAM subsystem," in *MICRO*, 2009.
- [37] Y. Ishii *et al.*, "High performance memory access scheduling using compute-phase prediction and writeback-refresh overlap," in *JJLP Memory Scheduling Championship*, 2012.
- [38] JEDEC, "DDR3 SDRAM Standard," 2010.
- [39] JEDEC, "DDR4 SDRAM Standard," 2012.
- [40] JEDEC, "Low Power Double Data Rate 3 (LPDDR3)," 2012.
- [41] JEDEC, "High Bandwidth Memory (HBM) DRAM," 2013.
- [42] JEDEC, "Low Power Double Data Rate 4 (LPDDR4)," 2014.

- [43] U. Kang *et al.*, "Co-Architecting Controllers and DRAM to Enhance DRAM Process Scaling," in *The Memory Forum*, 2014.
- [44] S. Khan *et al.*, "Detecting and Mitigating Data-Dependent DRAM Failures by Exploiting Current Memory Content," in *MICRO*, 2017.
- [45] S. Khan *et al.*, "A Case for Memory Content-Based Detection and Mitigation of Data-Dependent Failures in DRAM," *CAL*, 2016.
- [46] S. Khan *et al.*, "PARBOR: An Efficient System-Level Technique to Detect Data Dependent Failures in DRAM," in *DSN*, 2016.
- [47] S. Khan *et al.*, "The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study," in *SIGMETRICS*, 2014.
- [48] R. Kho *et al.*, "75nm 7Gb/s/pin 1Gb GDDR5 graphics memory device with bandwidth-improvement techniques," in *ISSCC*, 2011.
- [49] J. S. Kim *et al.*, "The DRAM Latency PUF: Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency-Reliability Tradeoff in Modern DRAM Devices," in *HPCA*, 2018.
- [50] J. Kim and M. C. Papaefthymiou, "Block-based multi-period refresh for energy efficient dynamic memory," in *ASIC*, 2001.
- [51] K. Kim and J. Lee, "A New Investigation of Data Retention Time in Truly Nanoscaled DRAMs," *EDL*, vol. 30, no. 8, pp. 846–848, 2009.
- [52] Y. Kim *et al.*, "Ramulator: A Fast and Extensible DRAM Simulator," *CAL*, 2015.
- [53] Y. Kim *et al.*, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," in *ISCA*, 2014.
- [54] Y. Kim *et al.*, "ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers," in *HPCA*, 2010.
- [55] Y. Kim *et al.*, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *MICRO*, 2010.
- [56] Y. Kim *et al.*, "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM," in *ISCA*, 2012.
- [57] Y. Kim *et al.*, "Exploiting the DRAM Microarchitecture to Increase Memory-Level Parallelism," *IPSI Transactions on Advanced Research (TAR)*, 2018.
- [58] T. Kirihaata *et al.*, "An 800-MHz embedded DRAM with a concurrent refresh mode," *IEEE JSSC*, pp. 1377–1387, 2005.
- [59] J. Kong *et al.*, "Towards Refresh-optimized EDRAM-based Caches with a Selective Fine-grain Round-robin Refresh Scheme," *Microprocess. Microsyst.*, vol. 49, no. C, pp. 95–104, 2017.
- [60] J. B. Kotra *et al.*, "Hardware-Software Co-design to Mitigate DRAM Refresh Overheads: A Case for Refresh-Aware Process Scheduling," in *ASPLOS*, 2017.
- [61] E. Kultursay *et al.*, "Evaluating STT-RAM as an energy-efficient main memory alternative," in *ISPASS*, 2013.
- [62] B. C. Lee *et al.*, "Architecting Phase Change Memory as a Scalable DRAM Alternative," in *ISCA*, 2009.
- [63] B. C. Lee *et al.*, "Phase Change Memory Architecture and the Quest for Scalability," *CACM*, vol. 53, no. 7, pp. 99–106, 2010.
- [64] B. C. Lee *et al.*, "Phase-Change Technology and the Future of Main Memory," *IEEE Micro*, vol. 30, no. 1, pp. 143–143, 2010.
- [65] C. J. Lee *et al.*, "DRAM-Aware Last-Level Cache Writeback: Reducing Write-Caused Interference in Memory Systems," Univ. of Texas at Austin, High Performance Systems Group, Tech. Rep. TR-HPS-2010-002, 2010.
- [66] C. J. Lee *et al.*, "Improving Memory Bank-Level Parallelism in the Presence of Prefetching," in *MICRO*, 2009.
- [67] D. Lee *et al.*, "Design-Induced Latency Variation in Modern DRAM Chips: Characterization, Analysis, and Latency Reduction Mechanisms," in *SIGMETRICS*, 2017.
- [68] D. Lee *et al.*, "Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM," in *PACT*, 2015.
- [69] D. Lee *et al.*, "Adaptive-Latency DRAM: Optimizing DRAM Timing for the Common-Case," in *HPCA*, 2015.
- [70] D. Lee *et al.*, "Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture," in *HPCA*, 2013.
- [71] D. Lee *et al.*, "Simultaneous Multi Layer Access: A High Bandwidth and Low Cost 3D-Stacked Memory Interface," *TACO*, 2016.
- [72] C. H. Lin *et al.*, "SECRET: Selective Error Correction for Refresh Energy Reduction in DRAMs," in *JCCD*, 2012.
- [73] J. Liu *et al.*, "An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms," in *ISCA*, 2013.
- [74] J. Liu *et al.*, "RAIDR: Retention-Aware Intelligent DRAM Refresh," in *ISCA*, 2012.
- [75] S. Liu *et al.*, "Flikker: Saving dram refresh-power through critical data partitioning," in *ASPLOS*, 2011.
- [76] S.-L. Lu *et al.*, "Improving DRAM Latency with Dynamic Asymmetric Subarray," in *MICRO*, 2015.
- [77] C.-K. Luk *et al.*, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *PLDI*, 2005.
- [78] Y. Luo *et al.*, "WARM: Improving NAND flash memory lifetime with write-hotness aware retention management," in *MSST*, 2015.
- [79] Y. Luo *et al.*, "Enabling Accurate and Practical Online Flash Channel Modeling for Modern MLC NAND Flash Memory," *JSAC*, 2016.
- [80] Y. Luo *et al.*, "HeatWatch: Improving 3D NAND Flash Memory Device Reliability by Exploiting Self-Recovery and Temperature Awareness," in *HPCA*, 2018.
- [81] Y. Luo *et al.*, "Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-Reliability Memory," in *DSN*, 2014.
- [82] J. A. Mandelman *et al.*, "Challenges and Future Directions for the Scaling of Dynamic Random-access Memory (DRAM)," *IBM JRD*, vol. 46, no. 2-3, pp. 187–212, 2002.
- [83] J. D. McCalpin, "STREAM Benchmark," <http://www.cs.virginia.edu/stream/>.
- [84] J. Meza *et al.*, "A Case for Efficient Hardware/Software Cooperative Management of Storage and Memory," in *WEED*, 2013.
- [85] J. Meza *et al.*, "Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field," in *DSN*, 2015.
- [86] Micron Technology, "8Gb: x4, x8 1.5V TwinDie DDR3 SDRAM," 2011.
- [87] Y. Moon *et al.*, "1.2V 1.6Gb/s 56nm 6F 2 4Gb DDR3 SDRAM with hybrid-I/O sense amplifier and segmented sub-array architecture," in *ISSCC*, 2009.
- [88] J. Mukundan *et al.*, "Understanding and mitigating refresh overheads in high-density DDR4 DRAM systems," in *ISCA*, 2013.
- [89] O. Mutlu, "The RowHammer problem and other issues we may face as memory becomes denser," in *DATE*, 2017.
- [90] O. Mutlu, "Memory Scaling: A Systems Architecture Perspective," *IMW*, 2013.
- [91] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems," in *ISCA*, 2008.
- [92] H. Naeimi *et al.*, "STT-RAM Scaling and Retention Failure," *Intel Technology Journal*, 2013.
- [93] P. Nair *et al.*, "A case for refresh pausing in DRAM memory systems," in *HPCA*, 2013.
- [94] P. J. Nair *et al.*, "ArchShield: Architectural Framework for Assisting DRAM Scaling by Tolerating High Error Rates," in *ISCA*, 2013.
- [95] T. Ohsawa *et al.*, "Optimizing the DRAM Refresh Count for Merged DRAM/Logic LSIs," in *ISLPED*, 1998.
- [96] M. Patel *et al.*, "The Reach Profiler (REAPER): Enabling the Mitigation of DRAM Retention Failures via Profiling at Aggressive Conditions," in *ISCA*, 2017.
- [97] A. Pirovano *et al.*, "Low-Field Amorphous State Resistance and Threshold Voltage Drift in Chalcogenide Materials," *TED*, 2004.
- [98] M. K. Qureshi *et al.*, "AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems," in *DSN*, 2015.
- [99] M. K. Qureshi *et al.*, "Enhancing Lifetime and Security of PCM-based Main Memory with Start-gap Wear Leveling," in *MICRO*, 2009.
- [100] M. K. Qureshi *et al.*, "Scalable High Performance Main Memory System Using Phase-change Memory Technology," in *ISCA*, 2009.
- [101] Rambus, "DRAM Power Model," 2010.
- [102] S. Rixner *et al.*, "Memory Access Scheduling," in *ISCA*, 2000.
- [103] SAFARI Research Group, "Ramulator – GitHub Repository," <https://github.com/CMU-SAFARI/ramulator>.
- [104] B. Schroeder *et al.*, "DRAM Errors in the Wild: A Large-Scale Field Study," in *SIGMETRICS*, 2009.
- [105] V. Seshadri *et al.*, "The Dirty-Block Index," in *ISCA*, 2014.
- [106] V. Seshadri *et al.*, "Fast Bulk Bitwise AND and OR in DRAM," *CAL*, 2015.
- [107] V. Seshadri *et al.*, "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization," in *MICRO*, 2013.
- [108] V. Seshadri *et al.*, "Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology," in *MICRO*, 2017.
- [109] A. Snaveley and D. Tullsen, "Symbiotic Jobscheduling for a Simultaneous Multithreading Processor," in *ASPLOS*, 2000.
- [110] Y. H. Son *et al.*, "Reducing Memory Access Latency with Asymmetric DRAM Bank Organizations," in *ISCA*, 2013.
- [111] V. Sridharan *et al.*, "Memory Errors in Modern Systems: The Good, The Bad, and The Ugly," in *ASPLOS*, 2015.
- [112] V. Sridharan and D. Liberty, "A Study of DRAM Failures in the Field," in *SC*, 2012.
- [113] Standard Performance Evaluation Corp., "SPEC CPU2006 Benchmarks," <http://www.spec.org/cpu2006>.
- [114] D. B. Strukov *et al.*, "The Missing Memristor Found," *Nature*, 2008.
- [115] J. Stuecheli *et al.*, "Elastic refresh: Techniques to mitigate refresh penalties in high density memory," in *MICRO*, 2010.
- [116] J. Stuecheli *et al.*, "The virtual write queue: Coordinating DRAM and last-level cache policies," in *ISCA*, 2010.
- [117] V. K. Tavva *et al.*, "EFG: An Enhanced Fine Granularity Refresh Feature for High-Performance DDR4 DRAM Devices," *TACO*, vol. 11, no. 3, 2014.
- [118] Transaction Performance Processing Council, "TPC Benchmarks," <http://www.tpc.org/>.
- [119] R. Venkatesan *et al.*, "Retention-Aware Placement in DRAM (RAPID): Software Methods for Quasi-Non-Volatile DRAM," in *HPCA*, 2006.
- [120] T. Vogelsang, "Understanding the Energy Consumption of Dynamic Random Access Memories," in *MICRO*, 2010.
- [121] H.-S. P. Wong *et al.*, "Metal-Oxide RRAM," *Proc. IEEE*, 2012.
- [122] H.-S. P. Wong *et al.*, "Phase Change Memory," *Proc. IEEE*, 2010.
- [123] H. Yoon *et al.*, "Row Buffer Locality Aware Caching Policies for Hybrid Memories," in *ICCD*, 2012.
- [124] H. Yoon *et al.*, "Efficient Data Mapping and Buffering Techniques for Multilevel Cell Phase-Change Memories," *TACO*, vol. 11, no. 4, pp. 40:1–40:25, 2014.
- [125] J. Yue and Y. Zhu, "Exploiting Subarrays Inside a Bank to Improve Phase Change Memory Performance," in *DATE*, 2013.
- [126] T. Zhang *et al.*, "CREAM: A Concurrent-Refresh-Aware DRAM Memory Architecture," in *HPCA*, 2014.

Exploiting Row-Level Temporal Locality in DRAM to Reduce the Memory Access Latency

Hasan Hassan^{1,2,3}

Gennady Pekhimenko^{4,2}

Nandita Vijaykumar²

Vivek Seshadri^{5,2}

Donghyuk Lee^{6,2}

Oguz Ergin³

Onur Mutlu^{1,2}

¹ETH Zürich

²Carnegie Mellon University

³TOBB University of Economics & Technology

⁴University of Toronto

⁵Microsoft Research India

⁶NVIDIA Research

This paper summarizes the idea of ChargeCache, which was published in HPCA 2016 [51], and examines the work's significance and future potential. DRAM latency continues to be a critical bottleneck for system performance. In this work, we develop a low-cost mechanism, called ChargeCache, that enables faster access to recently-accessed rows in DRAM, with no modifications to DRAM chips. Our mechanism is based on the key observation that a recently-accessed row has more charge and thus the following access to the same row can be performed faster. To exploit this observation, we propose to track the addresses of recently-accessed rows in a table in the memory controller. If a later DRAM request hits in that table, the memory controller uses lower timing parameters, leading to reduced DRAM latency. Row addresses are removed from the table after a specified duration to ensure rows that have leaked too much charge are not accessed with lower latency. We evaluate ChargeCache on a wide variety of workloads and show that it provides significant performance and energy benefits for both single-core and multi-core systems.

1. Problem: DRAM Latency

DRAM technology is commonly used as the main memory of modern computer systems. This is because DRAM is at a more favorable point in the trade-off spectrum of density (cost-per-bit) and access latency compared to other technologies like SRAM or flash. However, commodity DRAM devices are heavily optimized to maximize cost-per-bit. In fact, the latency of commodity DRAM has not reduced significantly in the past two decades [23, 25, 80, 83, 84, 108].

The latency of DRAM is heavily dependent on the design of the DRAM chip architecture, specifically the length of a wire called *bitline*. A DRAM chip consists of millions of DRAM cells. Each cell is composed of a transistor-capacitor pair. To access data from a cell, DRAM uses a component called *sense amplifier*. Each cell is connected to a sense amplifier using a *bitline*. To amortize the large cost of the sense amplifier, hundreds of DRAM cells are connected to the same bitline [84]. A longer bitline leads to higher resistance and parasitic capacitance on the path between a DRAM cell and the sense amplifier. As a result, longer bitlines result in higher DRAM access latency [80, 83, 84, 136].

To mitigate the negative effects of long DRAM access latency, existing systems rely on several major approaches.

First, they employ large on-chip caches to exploit the temporal and spatial locality of memory accesses. However, cache capacity is limited by chip area. Even caches as large as tens of megabytes may not be effective for some applications due to very large working sets and memory access characteristics that are not amenable to caching [61, 90, 113, 117, 118]. Second, systems employ aggressive prefetching techniques to preload data from memory before it is needed [5, 28, 138]. However, prefetching is inefficient for many irregular access patterns and it increases the bandwidth requirements and interference in the memory system [36, 38, 39, 76, 131, 138]. Third, systems employ multithreading [86, 134, 145]. However, this approach increases contention in the memory system [32, 37, 98, 106] and does not aid single-thread performance [62, 144]. Fourth, systems exploit memory level parallelism [31, 47, 104, 106, 107]. The DRAM architecture provides various levels of parallelism that can be exploited to simultaneously process multiple memory requests generated by modern processor architectures [78, 107, 115, 146]. While prior works [31, 33, 60, 78, 106, 112] propose techniques to better utilize the available parallelism, the benefits of these techniques are limited due to 1) address dependencies between instructions in the programs [6, 40, 103], and 2) resource conflicts in the memory subsystem [73, 120]. Unfortunately, *none* of these four approaches *fundamentally* reduce memory latency at its *source* and the DRAM latency continues to be a performance bottleneck in many systems.

2. Existing Techniques That Reduce DRAM Latency

DRAM latency can be reduced using several techniques, all of which have their own specific shortcomings. One simple approach to reduce DRAM latency is to use shorter bitlines. In fact, some specialized DRAM chips [48, 96, 125] offer lower latency by using shorter bitlines compared to commodity DRAM chips. Unfortunately, such chips come at a significantly higher cost than chips that use long bitlines, as they reduce the overall density of the device because they require more sense amplifiers, which occupy significant area [84]. Therefore, such specialized chips are usually not desirable for systems that require high memory capacity [29]. Prior works have proposed several heterogeneous DRAM architectures (e.g., segmented bitlines [84], asymmetric bank organizati-

ons [136], mechanisms that exploit the inherent latency variation across cells [25, 82]) that divide DRAM into two regions: one with low latency, and another with slightly higher latency. Such schemes propose to map frequently accessed data to the low-latency region, thereby achieving lower average memory access latency. However, such schemes might require 1) non-negligible changes to the cost-sensitive DRAM design, 2) techniques to create or identify low-latency regions in DRAM, and/or 3) mechanisms to identify, map, and migrate frequently-accessed data to low-latency regions. As a result, even though they reduce the latency for some portions of the DRAM chip, they may not be easy to adopt.

3. Key Observations

In our HPCA 2016 paper [51], we make two major observations that motivate a new mechanism for reducing DRAM latency,

Charge Variation. The amount of charge in the DRAM cells of a row determines the required latency for a DRAM access to that row. If the amount of charge in the cell is low, the sense amplifier completes its operation in longer time. Therefore, DRAM access latency increases. A DRAM cell loses its charge over time and the charge is replenished by a refresh operation or an access to the row. The access latency of a cell whose charge has been replenished recently can thus be significantly lower than the access latency of a cell that has less charge. Our SPICE simulations show that the first read/write command can be issued 44% faster to a highly-charged DRAM row compared to a row with less charge (see Section 6.2 and our HPCA 2016 paper [51]).

Row-Level Temporal Locality. We find that, mainly due to DRAM bank conflicts [73, 120], many applications tend to access rows that were recently closed (i.e., closed within a very short time interval). We refer to this form of temporal locality where certain rows are frequently closed and re-opened as *Row-Level Temporal Locality (RLTL)*. An important outcome of this observation is that a DRAM row remains in a *highly-charged* state when accessed for the *second* time within a short interval after the prior access. This is because accessing the DRAM row inherently replenishes the charge within the DRAM cells (just like a refresh operation does) [26, 46, 87, 88, 109, 133].

We define t -RLTL of an application for a given time interval t as the fraction of row activations in which the activation occurs within the time interval t after a *previous* pre-charge to the same row. Figure 1 shows the average RLTL for single-core and eight-core workloads with five different time intervals (from 0.125ms to 32ms). Our detailed experimental methodology is described in Section 5 of our HPCA 2016 paper [51]. For single-core workloads, the average 1ms-RLTL is 83%. In other words, 83% of all the row activations occur within 1ms after the same row was previously pre-charged. Due to the additional bank conflicts incurred as the number of workloads executing increases, for eight-core

workloads, the average 1ms-RLTL is 89%, significantly higher than that for the single-core workloads. These results show that RLTL of both single-core and eight-core workloads is significantly high even for small values of t , motivating us to exploit RLTL (i.e., row-level temporal locality) to detect highly-charged DRAM rows.¹

Note that a major reason for the high row-level temporal locality is the occurrence of bank conflicts in the DRAM subsystem. We find that, due to the bank conflicts, a row is likely to be requested again soon after it is precharged due to an intervening request to the same bank.

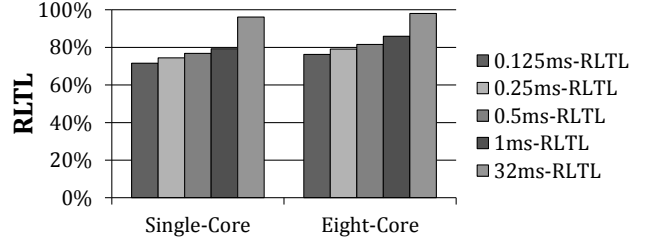


Figure 1: Average row-level temporal locality (RLTL) for 22 single-core and 20 eight-core workloads.

4. Our Goal

We observe that *many* applications exhibit high row-level temporal locality. In other words, for many applications, a significant fraction of the row activations occur within a small interval after the corresponding rows are precharged. As a result, such row activations can be served with lower activation latency than specified by the DRAM standard. **Our goal** in this work is to exploit this observation to reduce the effective DRAM access latency by tracking recently-accessed DRAM rows in the memory controller and reducing the latency for their next access(es). To this end, we propose an efficient mechanism, ChargeCache, which we describe in the next section.

5. Solution: ChargeCache

ChargeCache is based on three observations: 1) a row whose cells' charge has been recently replenished can be accessed with lower activation latency, 2) activating a row replenishes the charge on the cells of that row and the cells start leaking only after the *following* precharge command, and 3) many applications exhibit high row-level temporal locality, i.e., recently-activated rows are more likely to be activated again. Based on these observations, ChargeCache tracks rows that are recently activated, and serves near-future activations to such rows with lower latency by lowering the DRAM timing parameters for such activations.

As we show in Figure 2, ChargeCache adds a small table (structured as a cache), called *High-Charged Row Address Cache (HCRAC)*, to the memory controller that tracks the addresses of recently-accessed DRAM rows, i.e., highly-charged

¹For a more detailed study of row-level temporal locality, please see Section 3 of our HPCA 2016 paper [51].

rows. ChargeCache performs three operations. First, when a precharge command is issued to a bank, ChargeCache inserts the address of the row that was activated in the corresponding bank to the table (① in the figure). Second, when an activate command is issued, ChargeCache checks if the corresponding row address is present in the table (②). If the address is *not* present, then ChargeCache uses the standard DRAM timing parameters to issue subsequent commands to the bank. However, if the address of the activated row is present in the table, ChargeCache employs reduced timing parameters for subsequent commands to that bank. Our experimental results on multi-programmed applications show that, on average, ChargeCache can reduce the latency of 67% of all DRAM row activations (as shown in Section 6.4 of our HPCA 2016 paper [51]). Third, ChargeCache periodically invalidates old entries from the table to ensure that only rows that have sufficient amount of charge for being accessed with low latency remain in the table (③). Since a row may potentially reside in the table for very long time without being activated, such an operation is necessary to avoid a low-latency access to a row with small amount of charge (which could lead to wrong results).

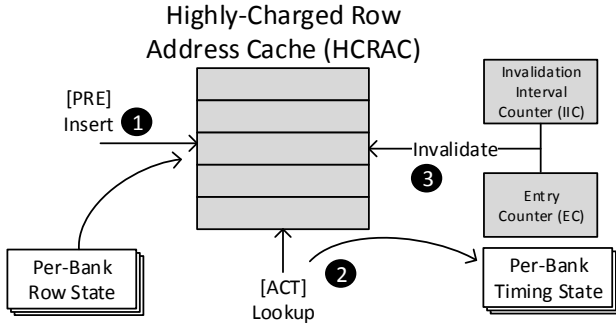


Figure 2: Components of the ChargeCache Mechanism. Reproduced from [51].

We name our mechanism *ChargeCache*, as it provides a *cache-like* benefit, i.e., latency reduction based on a locality property (i.e., RLTL), and does so by taking advantage of the *charge* level stored in a recently-activated row. The mechanism could potentially be used with current and emerging DRAM-based memories where the stored charge level leads to different access latencies. We release the source code of ChargeCache for two different versions of Ramulator [74, 122, 123] to enable future research to build upon our ideas.

6. Experimental Evaluation

In this section, we first explain our experimental methodology. Later, we quantitatively analyze the system performance improvement and DRAM energy savings that ChargeCache provides.

6.1. Methodology

We use circuit-level SPICE simulations to evaluate the DRAM latency reduction that can be achieved when accessing a highly-charged DRAM row. In Section 6.2, we show

the reduction in two DRAM timing parameters, t_{RCD} and t_{RAS} , that are affected by high charge amount stored in a DRAM cell.²

To evaluate the performance of ChargeCache, we use a cycle-accurate DRAM simulator, Ramulator [74, 122], in CPU-trace-driven mode. CPU traces are collected using a Pin-tool [91]. Table 1 lists the configuration of the evaluated systems. We implement the HCRAC table, which ChargeCache uses to store the addresses of recently accessed DRAM rows, similarly to a 2-way associative cache that uses the LRU policy.

Table 1: Simulated system configuration. Reproduced from [51].

Processor	1-8 cores, 4GHz clock frequency, 3-wide issue, 8 MSHRs/core, 128-entry instruction window
Last-level Cache	64B cache-line, 16-way associative, 4MB cache size
Memory Controller	64-entry read/write request queues, FR-FCFS scheduling policy [121, 153], open/closed row policy [71, 72] for single/multi core
DRAM	DDR3-1600 [97], 800MHz bus frequency, 1/2 channels, 1 rank/channel, 8 banks/rank, 64K rows/bank, 8KB row-buffer size, t_{RCD}/t_{RAS} 11/28 cycles
ChargeCache	128-entry (672 bytes)/core, 2-way associativity, LRU replacement policy, 1ms caching duration, t_{RCD}/t_{RAS} reduction 4/8 cycles

For area, power, and energy measurements, we modify McPAT [85] to implement ChargeCache using 22nm process technology. We use DRAMPower [22] to obtain power/energy results for the off-chip main memory subsystem. We feed DRAMPower with DRAM command traces obtained from our simulations using Ramulator.

We run 22 workloads from the SPEC CPU2006 [137], TPC [147], and STREAM [94] benchmark suites. We use SimPoint [50] to obtain traces from representative phases of each application. For single-core evaluations, unless stated otherwise, we run each workload for 1 billion instructions. For multi-core evaluations, we use 20 multiprogrammed workloads by assigning a randomly-chosen application to each core. We evaluate each configuration with its best-performing row-buffer management policy. Specifically, we use the open-row policy for single-core and closed-row policy for multi-core configurations. We simulate the benchmarks until each core

²For detail on DRAM timing parameters and operation, we refer the reader to our prior works [24, 25, 27, 51, 52, 68, 71, 72, 73, 74, 82, 83, 84, 87, 88, 114, 127, 128].

executes at least 1 billion instructions. For both single- and multi-core configurations, we first warm up the caches and ChargeCache by fast-forwarding 200 million cycles.

We measure performance improvement for single-core workloads using the Instructions per Cycle (IPC) metric. We measure multi-core performance using the weighted speedup [135] metric. Prior work has shown that weighted speedup is a measure of system-level job throughput [42].

6.2. Reduction in DRAM Timing Parameters

We evaluate the potential reduction in t_{RCD} and t_{RAS} for ChargeCache using circuit-level SPICE simulations. We implement the DRAM sense amplifier circuit using 55nm DDR3 model parameters [119] and PTM low-power transistor models [3, 152]. Figure 3 plots the variation in bitline voltage level during cell activation for different initial charge amounts of the cell.

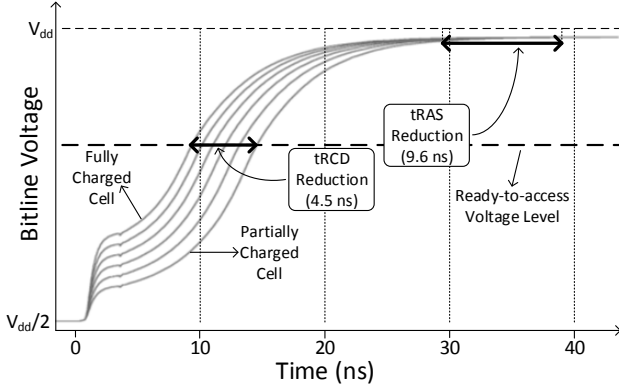


Figure 3: Effect of initial cell charge on bitline voltage. Reproduced from [51].

Depending on the initial charge (i.e., voltage level) of the cell, the bitline voltage increases at different speeds. When the cell is *fully-charged*, the sense amplifier is able to drive the bitline voltage to the *ready-to-access voltage level* in only 10ns. However, a partially-charged cell (i.e., one that has not been accessed for 64ms) brings the bitline voltage up slower. Specifically, the bitline connected to such a partially-charged cell reaches the ready-to-access voltage level in 14.5ns. Since DRAM timing parameters are dictated by this worst-case partially-charged state right before the refresh interval, we can achieve a 4.5ns reduction in t_{RCD} for a *fully-charged* cell. Similarly, the charge of the cell capacitor is restored at different times depending on the initial voltage of the cell. For a fully-charged cell, this results in a 9.6ns reduction in t_{RAS} .

In practice, we expect DRAM manufacturers to identify the lowered timing constraints for different caching durations. Today, DRAM manufacturers test each DRAM chip to determine if it meets the timing specifications. Similarly, we expect the manufacturers would also test each chip to determine if it meets the ChargeCache timing constraints.

6.3. Results

We experimentally evaluate the following mechanisms: 1) ChargeCache, 2) NUAT [133], which accesses *only* rows that are *recently-refreshed* at lower latency than the DRAM standard, 3) ChargeCache + NUAT, which is a combination of ChargeCache and NUAT [133] mechanisms, and 4) Low-Latency DRAM (LL-DRAM) [96], which is an idealized comparison point where we assume *all* rows in DRAM can be accessed with low latency, compared to our baseline DDR3-1600 memory, at any time, irrespective of when they are accessed or refreshed.

We compare the performance of our mechanism against the most closely related previous work, NUAT [133]. The key idea of NUAT is to access *recently-refreshed* rows at low latency, because these rows are already highly-charged. Thus, NUAT does *not* use low latency for rows that are *recently-accessed*, and hence it does *not* exploit the RLTL (Row-Level Temporal Locality) present in many applications.

Figure 4 shows the performance of single-core and eight-core workloads. The figure also includes the number of row misses per kilo-cycles (RMPKC) to show row activation intensity, which provides insight into the RLTL of the workload.

Single-Core Performance: Figure 4a shows the performance improvement over the baseline system for single-core workloads. These workloads are sorted in ascending order of RMPKC. ChargeCache achieves up to 9.3% (an average of 2.1%) speedup. Our mechanism outperforms NUAT and achieves a speedup close to LL-DRAM with a few exceptions. Applications that have a wide gap in performance between ChargeCache and LL-DRAM (e.g., *mcf*, *omnetpp*) access a large number of DRAM rows and exhibit high row-reuse distance [63]. A high row-reuse distance indicates that there is a large number of accesses to other rows between two accesses to the same row. Due to this reason, ChargeCache *cannot* retain the addresses of highly-charged rows until the next access to that row. Increasing the number of ChargeCache entries or employing cache management policies aware of reuse distance or thrashing [35, 117, 130, 148] may improve the performance of ChargeCache for such applications. We leave the evaluation of these methods for future work. We conclude that ChargeCache significantly reduces execution time for most high-RMPKC workloads and outperforms NUAT for all but few workloads.

Eight-Core Performance: Figure 4b shows the speedup on eight-core multiprogrammed workloads. On average, ChargeCache and NUAT improve performance by 8.6% and 2.5%, respectively. Employing ChargeCache in combination with NUAT achieves a 9.6% speedup, which is only 3.8% less than the improvement obtained using LL-DRAM. Although the multiprogrammed workloads are composed of the *same* applications as in single-core evaluations, we observe much higher performance improvements for the eight-core workloads. The reason is twofold. First, since multiple cores share a limited capacity LLC, simultaneously running applications compete

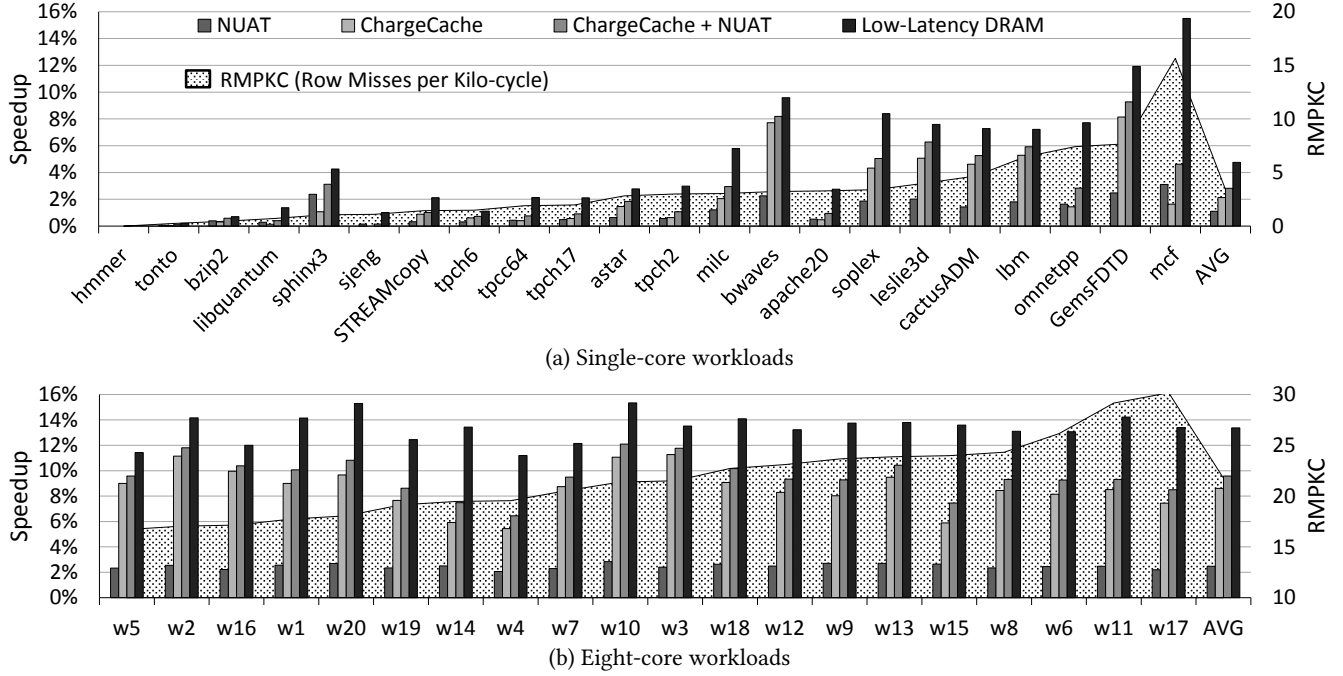


Figure 4: Speedup with ChargeCache, NUAT and Low-Latency DRAM for single-core and eight-core workloads. Reproduced from [51].

for the LLC. Thus, individual applications access main memory more often, which leads to higher RMPKC. This makes the workload performance more sensitive to main memory latency [20, 58, 73]. Second, the memory controllers receive memory requests from multiple simultaneously-running applications to a limited number of memory banks. Such requests are likely to target different rows since they use separate memory regions and these regions map to separate rows. Therefore, applications running concurrently exacerbate the bank-conflict rate and increase the number of row activations that hit in ChargeCache.

Overall, ChargeCache improves performance by up to 11.3% (8.1%) and on average 8.6% (2.1%) for eight-core (single-core) workloads. It outperforms NUAT for most of the applications. Using NUAT in combination with ChargeCache improves chsystem performance even further.

6.4. Impact on DRAM Energy

ChargeCache incurs negligible area and power overheads (see Section 6.5). Because it reduces execution time with negligible overhead, it leads to significant energy savings. Even though ChargeCache increases the energy efficiency of the entire system, we quantitatively evaluate the energy savings only for the DRAM subsystem since Ramulator [74] currently does not have a detailed CPU model. Figure 5 shows the average and maximum DRAM energy savings for single-core and eight-core workloads. ChargeCache reduces energy consumption by an average of 7.9% (1.8%), and by up to 14.1% (6.9%), for eight-core (single-core) workloads. We conclude that ChargeCache is effective at improving the energy efficiency of the DRAM subsystem, as well as the entire system.

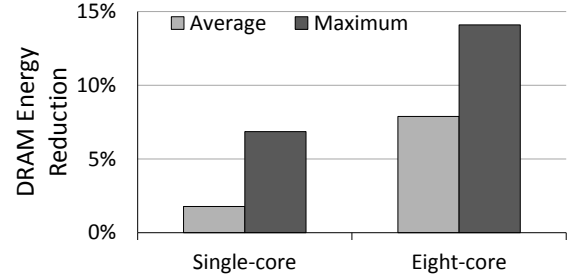


Figure 5: DRAM energy reduction of ChargeCache. Reproduced from [51].

6.5. Area and Power Consumption Overhead

HCRAC (Highly-Charged Row Address Cache) is the most area/power demanding component of ChargeCache. As we replicate HCRAC on a per-core and per-memory channel basis, the total area and power overhead ChargeCache introduces depends on the number of cores and memory channels.³ The total storage requirement is given by Equation 1, where C are MC are the number of cores and memory channels, respectively. LRU_{bits} depends on HCRAC associativity. $EntrySize$ is calculated using Equation 2, where R , B , and Ro are the number of ranks, banks, and rows in DRAM, respectively.

$$Storage_{bits} = C * MC * Entries * (EntrySize_{bits} + LRU_{bits}) \quad (1)$$

$$EntrySize_{bits} = \log_2(R) + \log_2(B) + \log_2(Ro) + 1 \quad (2)$$

Area. Our eight-core configuration has two memory channels. This introduces a total of 5376 bytes in storage requi-

³Note that sharing a single HCRAC across all or multiple cores can result in even lower overhead. We leave the exploration of such shared-HCRAC designs to future work.

rement for a 128-entry HCRAC, corresponding to an area of 0.022 mm^2 . This overhead is only 0.24% of the 4MB LLC.

Power Consumption. HCRAC is accessed on every *activate* and *precharge* command issued by the memory controller. On an *activate* command, HCRAC is searched for the corresponding row address. On a *precharge* command, the address of the precharged row is inserted into HCRAC. HCRAC entries are periodically invalidated to ensure they do not exceed a specified *caching duration*. These three operations increase dynamic power consumption in the memory controller, and the HCRAC storage increases static power consumption. Our analysis indicates that ChargeCache consumes 0.149 mW on average. This is only 0.23% of the average power consumption of the entire 4MB LLC. Note that we include the effect of this additional power consumption in our DRAM energy evaluations in Section 6.4. We conclude that ChargeCache incurs almost negligible chip area and power consumption overheads.

6.6. Other Results

We also evaluate and assess the sensitivity of ChargeCache benefits to ChargeCache capacity, caching duration, and temperature in Sections 6.4 and 7.1 of our HPCA 2016 paper [51].

7. Related Work

To our knowledge, this paper is the first to (i) show that applications typically exhibit significant *Row-level Temporal Locality (RLTL)* and (ii) exploit this locality to improve system performance by reducing the latency of requests to recently-accessed memory rows.

We have already (in Section 6.3) qualitatively and quantitatively compared ChargeCache to NUAT [133], which reduces access latency to *only* recently-refreshed rows. We have also shown that ChargeCache provides significantly higher average latency reduction than NUAT because RLTL is usually high, whereas the fraction of accesses to rows that are recently-refreshed is typically low (see Section 3 in our HPCA 2016 paper [51]).

Other previous works propose techniques to reduce performance degradation caused by long DRAM latencies. They focus on 1) enhancing the DRAM, 2) exploiting variations in manufacturing process and operating conditions, 3) developing various memory scheduling policies. We briefly summarize how ChargeCache differs from these works.

Enhancing DRAM Architecture. Lee et al. propose Tiered-Latency DRAM (TL-DRAM) [84], which divides each subarray into near and far segments using isolation transistors. With TL-DRAM, the memory controller accesses the near segment with lower latency since the isolation transistor reduces the bitline capacitance in that segment. Our mechanism could be implemented on top of TL-DRAM to reduce the access latency for both the near and far segment. Kim et al. propose SALP, which unlocks parallelism between subarrays at low cost, by modifying the DRAM chip

to enable pipelined access to subarrays [73]. The goal of SALP is to reduce the impact of bank conflicts by providing more parallelism and thereby reducing the latency of bank-conflict accesses. O et al. [110] propose a DRAM architecture where sense amplifiers are decoupled from bitlines to mitigate precharge latency. Choi et al. [30] propose to utilize multiple DRAM cells to store a single bit when sufficient DRAM capacity is available. By using multiple cells, they reduce activation, precharge and refresh latencies. Other works [24, 26, 49, 79, 126, 127, 128, 129, 136, 151] also propose new DRAM architectures to lower DRAM latency for various types of operations and accesses.

Processing-in-memory (PIM) architectures [1, 2, 8, 9, 34, 41, 44, 53, 54, 65, 69, 75, 111, 116, 127, 128, 129, 132, 139] using 3D-stacked memory [56, 59, 81, 89] reduce the *observed latency, from the perspective of the processor*, by moving some computation operations closer to DRAM. 3D-stacked memories are well suited for processing-in-memory due to their inclusion of a logic layer, which allows for the efficient implementation of CMOS logic in DRAM and offers high bandwidth to the DRAM layers. However, PIM architectures do not fundamentally reduce the access latency of the DRAM device, which ChargeCache does (for certain access patterns).

Unlike ChargeCache, a large number of these works require changes to the DRAM architecture itself. The approaches taken by these works are largely orthogonal to the ChargeCache approach and ChargeCache could be implemented together with any of these mechanisms to further reduce the DRAM latency.

Exploiting Process and Operating Condition Variations. Recent studies [21, 25, 27, 82, 83] propose methods to reduce the safety margins of the DRAM timing parameters when operating conditions are appropriate (i.e., not worst-case). Unlike these works, ChargeCache is largely independent of operating conditions like temperature, as discussed in Section 8.3, and is orthogonal to these latency reduction mechanisms.

Memory Request Scheduling Policies. Memory request scheduling policies (e.g., [4, 45, 55, 57, 66, 71, 72, 77, 99, 100, 105, 106, 121, 140, 141, 142, 143, 149, 153]) reduce the average DRAM access latency by improving DRAM parallelism, row buffer locality, and fairness in especially multi-core and heterogeneous systems. ChargeCache can be employed in conjunction with the scheduling policy that best suits the application and the underlying architecture.

8. Significance

Main memory latency has a critical impact on system performance [101]. Our work proposes a new low-cost mechanism to reduce DRAM latency, *without* any modifications to the existing DRAM chip architecture. In this section, we discuss the significance of our work by describing its novelty and expected long-term impact.

8.1. Novelty

ChargeCache reduces average DRAM latency by exploiting a type of DRAM access locality, *Row-Level Temporal Locality (RLTL)*, that commonly exists in workloads due to the presence of DRAM bank conflicts. Our work is the first to observe and formally define *RLTL* and exploit it to reduce DRAM latency by designing a new mechanism that takes advantage of *RLTL* and the fact that a DRAM row gets inherently refreshed on access. Our mechanism does not require any changes to the existing DRAM array structure of the DRAM chips and can be easily implemented on top of any DRAM standard with negligible overhead in the memory controller logic.

8.2. Applicability to Emerging DRAM Standards

ChargeCache is applicable to any memory technology where cells are volatile (leak charge over time) and the charge variation due to charge leakage has impact on access latency. ChargeCache can be used with a large set of standards derived from DDR (DDR_x, GDDR_x, LPDDR_x, etc.) [74] in a manner similar to the mechanism described in this work, without modifying the DRAM architecture. Using ChargeCache with 3D-stacked memories [81, 89] such as Wide I/O, HBM, and HMC [74] is also straightforward. The difference is that, for the technologies that implement the memory controller in the logic layer, the DRAM controller, and hence ChargeCache, can be easily implemented in the logic layer of the 3D-stacked memory chip instead of the processor chip.

We also believe that the key idea of ChargeCache is not limited to DRAM, and can potentially be applied to other memory technologies that store information in form of electrical charge, such as NAND flash memory [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 92, 93].

8.3. Long-Term Impact

8.3.1. Reducing DRAM Latency. During the last several decades, DRAM capacity increased significantly by shrinking the feature size of the transistors. Similarly, more efficient DRAM standards enabled memories with high bandwidth. The new 3D-stacking technology offers even higher bandwidth by incorporating DRAM and the logic layer on the same chip in a 3D-stacked manner. However, none of these advances lead to large improvements in the row access latency of the DRAM arrays. Hence, DRAM latency is already a critical bottleneck for system performance. Our work alleviates the DRAM latency problem with no overhead to the area-optimized DRAM chip, which is difficult to change, and with low overhead to the memory controller.

8.3.2. Row-Level Temporal Locality. Our paper is the first work to observe row-level temporal locality (*RLTL*). Note that *RLTL* is different from Row-Reuse Distance [63] that a prior work studies. Row-Reuse Distance is a metric indicating the number of accesses between two consecutive accesses to the same row. On the other hand, *RLTL* indicates the *time*

between two consecutive accesses to the same row. A row locality metric that includes time is important since charge leakage in DRAM is a function of *time*. In this work, we exploit *RLTL* to reduce DRAM latency. However, *RLTL* can also potentially be used to discover new techniques to improve different aspects of DRAM, such as reliability [70, 95, 101, 102] and bandwidth.

8.3.3. Importance for Future Systems. We believe the latency reduction mechanism of ChargeCache will become more important in future systems for four reasons. First, DRAM latency will become a much bigger bottleneck, as applications will become more data-intensive [101, 108]. Higher demand for data will result in more bank conflicts, as the number of DRAM banks is not scaling as fast as data intensity. Such applications will also have fast data access requirements, which will increase their sensitivity to the memory access latency [43, 64, 101, 108, 150]. As bank conflicts increase and accesses become more latency-critical, the benefits of ChargeCache will increase, as there will be higher *RLTL*, which ChargeCache can exploit to provide higher performance improvement.

Second, ChargeCache is likely to remain much more competitive than other state-of-the-art latency reduction techniques for the 3D-stacked memories of the future. These memories will likely operate at higher temperatures compared to conventional DRAM chips. The charge leakage rate of DRAM cells approximately doubles for every 10°C increase in temperature [67, 83, 87, 114]. This observation can be exploited to lower the DRAM latency when operating at low temperatures. A previous study, Adaptive-Latency DRAM (AL-DRAM) [83], proposes a mechanism to improve system performance by reducing the DRAM timing parameters at low operating temperature. AL-DRAM is based on the premise that DRAM typically does not operate at temperatures close to the worst-case temperature (85° C) even when it is heavily accessed. However, new 3D-stacked DRAM technologies such as HMC, HBM, WideIO may operate at significantly higher temperatures due to tight integration of multiple stack layers [7]. Therefore, state-of-the-art and compelling *dynamic latency scaling* techniques such as AL-DRAM may be less useful in these scenarios. In contrast to AL-DRAM, ChargeCache is *not* based on the charge difference that occurs due to temperature dependence. Rather, we exploit the high level of charge in recently-precharged rows to reduce timing parameters during later accesses to such rows. After conducting tests to determine the possible latency reduction in accessing highly-charged rows (for ChargeCache hits) at *worst-case* temperatures, we show that ChargeCache can be employed independently of the operating temperature (see Section 7.1 in our HPCA 2016 paper [51]).

Third, ChargeCache is complementary to other temperature-based and structural DRAM latency reduction techniques [24, 25, 73, 82, 84, 96, 133, 136]. ChargeCache can easily be used in conjunction with any of these techniques.

Fourth, ChargeCache is a low-cost mechanism, which does not require any changes to the existing DRAM chips, and requires only small changes to the memory controller. The low cost makes the adoption of ChargeCache more feasible in future systems than other proposed mechanisms, as these systems will be bottlenecked by power consumption, and thus by complexity [108].

Overall, we believe that ChargeCache will help to significantly reduce the memory access latency in future systems. To this end, to aid future research, we have released the source code of our ChargeCache simulator [123, 124] as part of our Ramulator releases [122, 123].

9. Conclusion

We introduce ChargeCache, a new, low-overhead mechanism that dynamically reduces the DRAM timing parameters for recently-accessed DRAM rows. ChargeCache exploits two key observations that we demonstrate in this work: 1) a recently-accessed DRAM row has cells with high amounts of charge and thus it can be accessed faster, and 2) many applications repeatedly access rows that are recently-accessed, due to bank conflicts.

Our extensive evaluations of ChargeCache on both single-core and multi-core systems show that it provides significant performance benefit and DRAM energy reduction at very modest hardware overhead. ChargeCache requires no modifications to the existing DRAM chips and occupies only a small area on the memory controller.

We conclude that ChargeCache is a simple yet efficient mechanism to dynamically reduce DRAM latency, which significantly improves both the performance and energy efficiency of modern systems. We hope that our observation of the phenomenon of row-level temporal locality and its simple exploitation to reduce DRAM latency inspires other works to develop other new techniques to improve memory subsystem characteristics like performance, efficiency, and reliability.

Acknowledgments

We thank Saugata Ghose for his dedicated effort in the preparation of this article. We thank the reviewers and the SAFARI group members for their feedback. We acknowledge the generous support of Google, Intel, NVIDIA, Samsung, and VMware. This work is supported in part by NSF grants 1212962, 1320531, and 1409723, the Intel Science and Technology Center for Cloud Computing, and the Semiconductor Research Corporation.

References

- [1] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing," in *ISCA*, 2015.
- [2] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture," in *ISCA*, 2015.
- [3] Arizona State Univ., NIMO Group, "Predictive Technology Model," <http://ptm.asu.edu/>.
- [4] R. Ausavarungnirun, K. K.-W. Chang, L. Subramanian, G. H. Loh, and O. Mutlu, "Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems," in *ISCA*, 2012.
- [5] J.-L. Baer and T.-F. Chen, "An Effective on-Chip Preloading Scheme to Reduce Data Access Penalty," in *ICS*, 1991.
- [6] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz, and U. Weiser, "Correlated Load-Address Predictors," in *ISCA*, 1999.
- [7] B. Black *et al.*, "Die Stacking (3D) Microarchitecture," in *MICRO*, 2006.
- [8] A. Boroumand *et al.*, "Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks," in *ASPLOS*, 2018.
- [9] A. Boroumand, S. Ghose, B. Lucia, K. Hsieh, K. Malladi, H. Zheng, and O. Mutlu, "LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory," *IEEE CAL*, 2017.
- [10] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, and O. Mutlu, "Error Characterization, Mitigation, and Recovery in Flash Memory Based Solid-State Drives," arXiv:1706.08642 [cs.AR], 2017.
- [11] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, and O. Mutlu, "Errors in Flash-Memory-Based Solid-State Drives: Analysis, Mitigation, and Recovery," arXiv:1711.11427 [cs.AR], 2017.
- [12] Y. Cai, S. Ghose, Y. Luo, K. Mai, O. Mutlu, and E. F. Haratsch, "Vulnerabilities in MLC NAND Flash Memory Programming: Experimental Analysis, Exploits, and Mitigation Techniques," in *HPCA*, 2017.
- [13] Y. Cai, E. F. Haratsch, O. Mutlu, and K. Mai, "Error Patterns in MLC NAND Flash Memory: Measurement, Characterization, and Analysis," in *DATE*, 2012.
- [14] Y. Cai, G. Yalcin, O. Mutlu, E. F. Haratsch, O. Unsal, A. Cristal, and K. Mai, "Neighbor Cell Assisted Error Correction in MLC NAND Flash Memories," in *SIGMETRICS*, 2014.
- [15] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, and O. Mutlu, "Error Characterization, Mitigation, and Recovery in Flash-Memory-Based Solid-State Drives," *Proc. IEEE*, 2017.
- [16] Y. Cai, E. F. Haratsch, O. Mutlu, and K. Mai, "Threshold Voltage Distribution in MLC NAND Flash Memory: Characterization, Analysis, and Modeling," in *DATE*, 2013.
- [17] Y. Cai, Y. Luo, S. Ghose, E. F. Haratsch, K. Mai, and O. Mutlu, "Read Disturb Errors in MLC NAND Flash Memory: Characterization, Mitigation, and Recovery," in *DSN*, 2015.
- [18] Y. Cai, Y. Luo, E. F. Haratsch, K. Mai, and O. Mutlu, "Data Retention in MLC NAND Flash Memory: Characterization, Optimization, and Recovery," in *HPCA*, IEEE, 2015.
- [19] Y. Cai, G. Yalcin, O. Mutlu, E. F. Haratsch, A. Cristal, O. S. Unsal, and K. Mai, "Flash Correct-and-Refresh: Retention-Aware Error Management for Increased Flash Memory Lifetime," in *ICCD*, 2012.
- [20] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture," in *HPCA*, 2005.
- [21] K. Chandrasekar, S. Goossens, C. Weis, M. Koedam, B. Akesson, N. Wehn, and K. Goossens, "Exploiting Expendable Process-Margins in DRAMs for Run-Time Performance Optimization," in *DATE*, 2014.
- [22] K. Chandrasekar, C. Weis, B. Akesson, N. Wehn, and K. Goossens, "Towards Variation-Aware System-Level Power Estimation of DRAMs: An Empirical Approach," in *DAC*, 2013.
- [23] K. K. Chang, "Understanding and Improving the Latency of DRAM-Based Memory Systems," Ph.D. dissertation, Carnegie Mellon Univ., 2017.
- [24] K. K. Chang *et al.*, "Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM," in *HPCA*, 2016.
- [25] K. K. Chang, A. Kashyap, H. Hassan, S. Ghose, K. Hsieh, D. Lee, T. Li, G. Pehimenko, S. Khan, and O. Mutlu, "Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization," in *SIGMETRICS*, 2016.
- [26] K. K. Chang, D. Lee, Z. Chishti, A. R. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu, "Improving DRAM Performance by Parallelizing Refreshes with Accesses," in *HPCA*, 2014.
- [27] K. K. Chang, A. G. Yaalikci, S. Ghose, A. Agrawal, N. Chatterjee, A. Kashyap, D. Lee, M. O'Connor, H. Hassan, and O. Mutlu, "Understanding Reduced-Voltage Operation in Modern DRAM Devices: Experimental Characterization, Analysis, and Mechanisms," *SIGMETRICS*, 2017.
- [28] M. J. Charney and T. R. Puzak, "Prefetching and Memory System Behavior of the SPEC95 Benchmark Suite," *IBM JRD*, 1997.
- [29] N. Chatterjee, M. Shevgoor, R. Balasubramanian, A. Davis, Z. Fang, R. Illikkal, and R. Iyer, "Leveraging Heterogeneity in DRAM Main Memories to Accelerate Critical Word Access," in *MICRO*, 2012.
- [30] J. Choi, W. Shin, J. Jang, J. Suh, Y. Kwon, Y. Moon, and L.-S. Kim, "Multiple Clone Row DRAM: A Low Latency and Area Optimized DRAM," in *ISCA*, 2015.
- [31] Y. Chou, B. Fahs, and S. Abraham, "Microarchitecture Optimizations for Exploiting Memory-Level Parallelism," in *ISCA*, 2004.
- [32] R. Das, R. Ausavarungnirun, O. Mutlu, A. Kumar, and M. Azimi, "Application-to-Core Mapping Policies to Reduce Memory System Interference in Multi-Core Systems," in *HPCA*, 2013.
- [33] W. Ding, D. Guttman, and M. Kandemir, "Compiler Support for Optimizing Memory Bank-Level Parallelism," in *MICRO*, 2014.
- [34] J. Draper *et al.*, "The Architecture of the DIVA Processing-in-Memory Chip," in *ICS*, 2002.
- [35] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum, "Improving Cache Management Policies Using Dynamic Reuse Distances," in *MICRO*, 2012.

- [36] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Prefetch-Aware Shared Resource Management for Multi-Core Systems," in *ISCA*, 2011.
- [37] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt, "Parallel Application Memory Scheduling," in *MICRO*, 2011.
- [38] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, "Coordinated Control of Multiple Prefetchers in Multi-Core Systems," in *MICRO*, 2009.
- [39] E. Ebrahimi, O. Mutlu, and Y. N. Patt, "Techniques for Bandwidth-Efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems," in *HPCA*, 2009.
- [40] R. J. Eickemeyer and S. Vassiliadis, "A Load-Instruction Unit for Pipelined Processors," *IBM JRD*, 1993.
- [41] D. G. Elliott, M. Stumm, W. M. Snelgrove, C. Cocjaru, and R. McKenzie, "Computational RAM: Implementing Processors in Memory," *IEEE DT*, 1999.
- [42] S. Eyerhan and L. Eeckhout, "System-Level Performance Metrics for Multiprogram Workloads," *IEEE Micro*, 2008.
- [43] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafae, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the Clouds: a Study of Emerging Scale-Out Workloads on Modern Hardware," in *ASPLOS*, 2012.
- [44] B. B. Fraguera, J. Renau, P. Feautrier, D. Padua, and J. Torrellas, "Programming the FlexRAM Parallel Intelligent Memory System," in *PPoPP*, 2003.
- [45] S. Ghose, H. Lee, and J. F. Martinez, "Improving Memory Scheduling via Processor-Side Load Criticality Information," in *ISCA*, 2013.
- [46] M. Ghosh and H.-H. S. Lee, "Smart Refresh: An Enhanced Memory Controller Design for Reducing Energy in Conventional and 3D Die-Stacked DRAMs," in *MICRO*, 2007.
- [47] A. Glew, "MLP yes! ILP no," in *ASPLOS WACI*, 1998.
- [48] GSI, "Low Latency DRAMs," <http://www.gsistechnology.com>.
- [49] N. D. Guler, R. Manikantan, M. Mehendale, and R. Govindarajan, "Multiple Sub-Row Buffers in DRAM: Unlocking Performance and Energy Improvement Opportunities," in *ICS*, 2012.
- [50] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and More Flexible Program Phase Analysis," *JILP*, 2005.
- [51] H. Hassan, G. Pekhimenko, N. Vijaykumar, V. Seshadri, D. Lee, O. Ergin, and O. Mutlu, "ChargeCache: Reducing DRAM Latency by Exploiting Row Access Locality," in *HPCA*, 2016.
- [52] H. Hassan, N. Vijaykumar, S. Khan, S. Ghose, K. Chang, G. Pekhimenko, D. Lee, O. Ergin, and O. Mutlu, "SoftMC: A Flexible and Practical Open-Source Infrastructure for Enabling Experimental DRAM Studies," in *HPCA*, 2017.
- [53] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler, "Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems," in *ISCA*, 2016.
- [54] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu, "Accelerating Pointer Chasing in 3D-Stacked Memory: Challenges, Mechanisms, Evaluation," in *ICCD*, 2016.
- [55] I. Hur and C. Lin, "Adaptive History-Based Memory Schedulers," in *MICRO*, 2004.
- [56] Hybrid Memory Cube Consortium, "Hybrid Memory Cube specification 2.0," November 2014.
- [57] E. Ipek, O. Mutlu, J. F. Martinez, and R. Caruana, "Self-Optimizing Memory Controllers: A Reinforcement Learning Approach," in *ISCA*, 2008.
- [58] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt, "QoS Policies and Architecture for Cache/Memory in CMP Platforms," in *SIGMETRICS*, 2007.
- [59] JEDEC, "High Bandwidth Memory (HBM) DRAM," 2013.
- [60] M. K. Jeong, D. H. Yoon, D. Sunwoo, M. Sullivan, I. Lee, and M. Erez, "Balancing DRAM Locality and Parallelism in Shared Memory CMP Systems," in *HPCA*, 2012.
- [61] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi, "Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache," in *MICRO*, 2014.
- [62] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, "Bottleneck Identification and Scheduling in Multithreaded Applications," in *ASPLOS*, 2012.
- [63] M. Kandemir, H. Zhao, X. Tang, and M. Karakoy, "Memory Row Reuse Distance and its Role in Optimizing Application Performance," in *SIGMETRICS*, 2015.
- [64] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a Warehouse-Scale Computer," in *ISCA*, 2015.
- [65] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas, "FlexRAM: Toward an Advanced Intelligent Memory System," in *ICCD*, 2012.
- [66] D. Kaseridis, J. Stuecheli, and L. K. John, "Minimalist Open-Page: A DRAM Page-Mode Scheduling Policy for the Many-Core Era," in *MICRO*, 2011.
- [67] S. Khan, D. Lee, Y. Kim, A. R. Alameldeen, C. Wilkerson, and O. Mutlu, "The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study," in *SIGMETRICS*, 2014.
- [68] J. S. Kim, M. Patel, H. Hassan, and O. Mutlu, "The DRAM Latency PUF: Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency-Reliability Tradeoff in Modern DRAM Devices," in *HPCA*, 2018.
- [69] J. S. Kim, D. Senol, H. Xin, D. Lee, S. Ghose, M. Alser, H. Hassan, O. Ergin, C. Alkan, and O. Mutlu, "GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-in-Memory Technologies," *BMC Genomics*, 2018.
- [70] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping Bits in Memory without Accessing Them: An Experimental Study of DRAM Disturbance Errors," in *ISCA*, 2014.
- [71] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers," in *HPCA*, 2010.
- [72] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *MICRO*, 2010.
- [73] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM," in *ISCA*, 2012.
- [74] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," in *CAL*, 2015.
- [75] P. M. Kogge, "EXECUBE-a New Architecture for Scaleable MPPs," in *ICPP*, 1994.
- [76] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt, "Prefetch-Aware DRAM Controllers," in *MICRO*, 2008.
- [77] 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," *UT-Austin, HPS, Tech. Report*, 2010.
- [78] C. J. Lee, V. Narasiman, O. Mutlu, and Y. N. Patt, "Improving Memory Bank-Level Parallelism in the Presence of refetching," in *MICRO*, 2009.
- [79] D. Lee, L. Subramanian, R. Ausavarungnirun, J. Choi, and O. Mutlu, "Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM," in *PACT*, 2015.
- [80] D. Lee, "Reducing DRAM Latency at Low Cost by Exploiting Heterogeneity," Ph.D. dissertation, Carnegie Mellon Univ., 2016.
- [81] D. Lee, S. Ghose, G. Pekhimenko, S. Khan, and O. Mutlu, "Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost," *TACO*, 2016.
- [82] D. Lee, S. Khan, L. Subramanian, S. Ghose, R. Ausavarungnirun, G. Pekhimenko, V. Seshadri, and O. Mutlu, "Design-Induced Latency Variation in Modern DRAM Chips: Characterization, Analysis, and Latency Reduction Mechanisms," *SIGMETRICS*, 2017.
- [83] D. Lee, Y. Kim, G. Pekhimenko, S. Khan, V. Seshadri, K. Chang, and O. Mutlu, "Adaptive-Latency DRAM: Optimizing DRAM Timing for the Common-Case," in *HPCA*, 2015.
- [84] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu, "Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture," in *HPCA*, 2013.
- [85] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *MICRO*, 2009.
- [86] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro*, 2008.
- [87] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu, "An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms," in *ISCA*, 2013.
- [88] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, "RAIDR: Retention-Aware Intelligent DRAM Refresh," in *ISCA*, 2012.
- [89] G. H. Loh, "3D-Stacked Memory Architectures for Multi-Core Processors," in *ISCA*, 2008.
- [90] P. Lotfi-Kamran *et al.*, "Scale-Out Processors," in *ISCA*, 2012.
- [91] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *PLDI*, 2005.
- [92] Y. Luo, S. Ghose, Y. Cai, E. F. Haratsch, and O. Mutlu, "Enabling Accurate and Practical Online Flash Channel Modeling for Modern MLC NAND Flash Memory," *JSAC*, 2016.
- [93] Y. Luo, S. Ghose, Y. Cai, E. F. Haratsch, and O. Mutlu, "HeatWatch: Improving 3D NAND Flash Memory Device Reliability by Exploiting Self-Recovery and Temperature Awareness," in *HPCA*, 2018.
- [94] J. D. McCalpin, "STREAM Benchmark," <http://www.streambench.org/>.
- [95] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, "Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field," in *DSN*, 2015.
- [96] Micron, "RLDRAM Memory," <http://www.micron.com/products/dram/rlDRAM-memory>.
- [97] Micron Technology, "4Gb: x4, x8, x16 DDR3 SDRAM," 2011.
- [98] T. Moscibroda and O. Mutlu, "Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems," in *USENIX Security*, 2007.
- [99] T. Moscibroda and O. Mutlu, "Distributed Order Scheduling and its Application to Multi-Core DRAM Controllers," in *PODC*, 2008.
- [100] J. Mukundan and J. F. Martinez, "MORSE: Multi-Objective Reconfigurable Self-Optimizing Memory Scheduler," in *HPCA*, 2012.
- [101] O. Mutlu, "Memory Scaling: A Systems Architecture Perspective," in *IMW*, 2013.
- [102] O. Mutlu, "The RowHammer Problem and Other Issues We May Face as Memory Becomes Denser," in *DATE*, 2017.
- [103] O. Mutlu, H. Kim, and Y. N. Patt, "Address-Value Delta (AVD) Prediction: Increasing the Effectiveness of Runahead Execution by Exploiting Regular Memory Allocation Patterns," in *MICRO*, 2005.
- [104] O. Mutlu, H. Kim, and Y. N. Patt, "Techniques for Efficient Processing in Runahead Execution Engines," in *ISCA*, 2005.
- [105] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *MICRO*, 2007.

- [106] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems," in *ISCA*, 2008.
- [107] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors," in *HPCA*, 2003.
- [108] O. Mutlu and L. Subramanian, "Research Problems and Opportunities in Memory Systems," *SUPERFRI*, 2014.
- [109] P. Nair, C.-C. Chou, and M. K. Qureshi, "A Case for Refresh Pausing in DRAM Memory Systems," in *HPCA*, 2013.
- [110] S. O. Y. H. Son, N. S. Kim, and J. H. Ahn, "Row-Buffer Decoupling: A Case for Low-Latency DRAM Microarchitecture," in *ISCA*, 2014.
- [111] M. Oskin, F. T. Chong, and T. Sherwood, "Active Pages: A Computation Model for Intelligent Memory," in *ISCA*, 1998.
- [112] V. S. Pai and S. Adve, "Code Transformations to Improve Memory Parallelism," in *MICRO*, 1999.
- [113] S. Palacharla and R. Kessler, "Evaluating Stream Buffers as a Secondary Cache Replacement," in *ISCA*, 1994.
- [114] M. Patel, J. S. Kim, and O. Mutlu, "The Reach Profiler (REAPER): Enabling the Mitigation of DRAM Retention Failures via Profiling at Aggressive Conditions," in *ISCA*, 2017.
- [115] Y. N. Patt, W.-m. Hwu, and M. Shebanow, "HPS, A New Microarchitecture: Rationale and Introduction," in *MICRO*, 1985.
- [116] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A Case for Intelligent RAM," *IEEE Micro*, 1997.
- [117] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive Insertion Policies for High Performance Caching," in *ISCA*, 2007.
- [118] M. K. Qureshi, M. A. Suleman, and Y. N. Patt, "Line Distillation: Increasing Cache Capacity by Filtering Unused Words in Cache Lines," in *HPCA*, 2007.
- [119] Rambus, "DRAM Power Model (2010)," <http://www.rambus.com/energy>.
- [120] B. R. Rau, "Pseudo-Randomly Interleaved Memory," in *ISCA*, 1991.
- [121] S. Rixner, J. D. Owens, P. Mattson, U. J. Kapasi, and W. J. Dally, "Memory Access Scheduling," in *ISCA*, 2000.
- [122] SAFARI Research Group, "Ramulator - GitHub Repository," <https://github.com/CMU-SAFARI/ramulator>.
- [123] SAFARI Research Group, "RamulatorSharp - GitHub Repository," <https://github.com/CMU-SAFARI/RamulatorSharp>.
- [124] SAFARI Research Group, "SAFARI Software Tools - GitHub Repository," <https://github.com/CMU-SAFARI>.
- [125] Y. Sato *et al.*, "Fast Cycle RAM (FCRAM): A 20-ns Random Row Access, Pipelined Operating DRAM," in *VLSI Circuits*, 1998.
- [126] V. Seshadri, K. Hsieh, A. Boroumand, D. Lee *et al.*, "Fast Bulk Bitwise AND and OR in DRAM," in *CAL*, 2015.
- [127] V. Seshadri *et al.*, "RowClone: Fast and Energy-Efficient in-DRAM Bulk Data Copy and Initialization," in *MICRO*, 2013.
- [128] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology," in *MICRO*, 2017.
- [129] V. Seshadri, T. Mullins, A. Boroumand, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Gather-Scatter DRAM: In-DRAM Address Translation to Improve the Spatial Locality of Non-Unit Strided Accesses," in *MICRO*, 2015.
- [130] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry, "The Evicted-Address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing," in *PACT*, 2012.
- [131] V. Seshadri, S. Yedkar, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Mitigating Prefetcher-Caused Pollution Using Informed Caching Policies for Prefetched Blocks," *TACO*, 2015.
- [132] D. E. Shaw, S. Stolfo, H. Ibrahim, B. K. Hillyer, J. Andrews, and G. Wiederhold, "The NON-VON Database Machine: An Overview," Columbia Univ. Dept. of Computer Science, Tech. Rep. CUCS-022-81, 1981.
- [133] W. Shin, J. Yang, J. Choi, and L.-S. Kim, "NUAT: A Non-Uniform Access Time Memory Controller," in *HPCA*, 2014.
- [134] B. J. Smith, "A Pipelined, Shared Resource MIMD Computer," in *ICPP*, 1978.
- [135] A. Snively and D. M. Tullsen, "Symbiotic Jobscheduling for a Simultaneous Multithreading Processor," *ASPLOS*, 2000.
- [136] Y. H. Son, O. Seongil, Y. Ro, J. W. Lee, and J. H. Ahn, "Reducing Memory Access Latency with Asymmetric DRAM Bank Organizations," in *ISCA*, 2013.
- [137] SPEC CPU2006, "Standard Performance Evaluation Corporation," <http://www.spec.org/cpu2006>.
- [138] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers," in *HPCA*, 2007.
- [139] H. S. Stone, "A Logic-in-Memory Computer," *IEEE Trans. Comput.*, 1970.
- [140] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, "The Blacklisting Memory Scheduler: Achieving High Performance and Fairness at Low Cost," in *ICCD*, 2014.
- [141] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, "The Blacklisting Memory Scheduler: Balancing Performance, Fairness and Complexity," *TPDS*, 2015.
- [142] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu, "The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory," in *MICRO*, 2015.
- [143] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu, "MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems," in *HPCA*, 2013.
- [144] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, "Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures," in *ASPLOS*, 2009.
- [145] J. E. Thornton, "Parallel Operation in the Control Data 6600," in *Fall Joint Computer Conference*, 1964.
- [146] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM JRD*, 1967.
- [147] Transaction Processing Performance Council, "TPC Benchmarks," <http://www.tpc.org/>.
- [148] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun, "A Modified Approach to Data Cache Management," in *MICRO*, 1995.
- [149] H. Usui, L. Subramanian, K. K.-W. Chang, and O. Mutlu, "DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators," *TACO*, 2016.
- [150] L. Wang *et al.*, "BigDataBench: A Big Data Benchmark Suite from Internet Services," in *HPCA*, 2014.
- [151] T. Zhang, K. Chen, C. Xu, G. Sun, T. Wang, and Y. Xie, "Half-DRAM: A High-Bandwidth and Low-Power DRAM Architecture from the Rethinking of Fine-Grained Activation," in *ISCA*, 2014.
- [152] W. Zhao and Y. Cao, "New Generation of Predictive Technology Model for Sub-45 nm Early Design Exploration," *IEEE TED*, 2006.
- [153] W. K. Zuravleff and T. Robinson, "Controller for a Synchronous DRAM that Maximizes Throughput by Allowing Memory Requests and Commands to be Issued Out of Order," US Patent 5,630,096. 1997.

Heterogeneous-Reliability Memory: Exploiting Application-Level Memory Error Tolerance

Yixin Luo¹ Sriram Govindan² Bikash Sharma^{3,2} Mark Santaniello^{3,2} Justin Meza^{3,1}
Aman Kansal² Jie Liu² Badriddine Khessib² Kushagra Vaid² Onur Mutlu^{4,1}

¹Carnegie Mellon University ²Microsoft Corporation ³Facebook ⁴ETH Zürich

This paper summarizes our work on characterizing application memory error vulnerability to optimize datacenter cost via Heterogeneous-Reliability Memory (HRM), which was published in DSN 2014 [104], and examines the work’s significance and future potential. Memory devices represent a key component of datacenter total cost of ownership (TCO), and techniques used to reduce errors that occur on these devices increase this cost. Existing approaches to providing reliability for memory devices pessimistically treat all data as equally vulnerable to memory errors. Our key insight is that there exists a diverse spectrum of tolerance to memory errors in new data-intensive applications, and that traditional one-size-fits-all memory reliability techniques are inefficient in terms of cost. For example, we found that while traditional error protection increases memory system cost by 12.5%, some applications can achieve 99.00% availability on a single server with a large number of memory errors without any error protection. This presents an opportunity to greatly reduce server hardware cost by provisioning the right amount of memory reliability for different applications.

Toward this end, in our DSN 2014 paper [104], we make three main contributions to enable highly-reliable servers at low datacenter cost. First, we develop a new methodology to quantify the tolerance of applications to memory errors. Second, using our methodology, we perform a case study of three new data-intensive workloads (an interactive web search application, an in-memory key-value store, and a graph mining framework) to identify new insights into the nature of application memory error vulnerability. Third, based on our insights, we propose several new hardware/software heterogeneous-reliability memory system designs to lower datacenter cost while achieving high reliability and discuss their trade-offs. We show that our new techniques can reduce server hardware cost by 4.7% while achieving 99.90% single server availability.

We believe the notion of HRM opens up a sea of opportunities in optimizing memory system and overall system cost, reliability, efficiency, and performance in a manner that is aware of applications’ tolerance to memory errors. Thus, our paper just scratches the surface of a large HRM exploration space, which we hope future works will undertake in various novel ways, in a wide variety of systems, ranging from datacenters to mobile and embedded systems.

1. Introduction

A warehouse-scale datacenter consists of many thousands of machines running a diverse set of applications, and com-

prises the foundation of the modern web [4, 151]. While such datacenters are vital to the operation of companies such as Facebook, Google, Microsoft, and Yahoo!, reducing the cost of such large-scale deployments of machines poses a significant challenge to these and other companies. Recently, the need for reduced datacenter cost has driven companies to examine more energy-efficient server designs [38] and build their datacenter installations in cold environments to reduce cooling costs [49, 59] or use built-in power plants to reduce electricity supply costs [139].

There are two main components of the *total cost of ownership* (TCO) of a datacenter [4]: (1) capital costs (those associated with server hardware) and (2) operational costs (those associated with providing electricity and cooling). Recent studies have shown that capital costs can account for the majority (e.g., around 57% in [4]) of datacenter TCO, and thus represent the main impediment for reducing datacenter TCO. In addition, this component of datacenter TCO is only expected to increase going forward as companies adopt more efficient cooling and power supply techniques.

Of the dominant component of datacenter TCO (capital costs associated with server hardware), the cost of server processors and memory represents the key component—around 60% in modern servers [77]. Furthermore, the cost of the memory in today’s servers is comparable to that of the processors [77], and is likely to exceed processor cost for data-intensive applications such as web search and social media services, which use in-memory caching to improve response time [54, 127, 128, 129, 159] (e.g., a popular key-value store, Memcached, has been used at Google and Facebook [54, 127] for this purpose).

Exacerbating the cost of memory in modern servers is the use of memory devices (such as dynamic random access memory, or DRAM) that provide error detection and correction. This cost arises from two components: (1) quality assurance testing performed by memory vendors to ensure devices sold to customers are of a high enough caliber and (2) additional memory capacity for error detection and correction. Device testing has been shown to account for an increasing fraction of the cost of memory for DRAM [2, 33]. The cost of additional memory capacity, on the other hand, depends on the technique used to provide error detection and correction.

Table 1 compares several common memory error detection and correction techniques in terms of which types of errors

they are able to detect/correct and the additional amount of capacity/logic they require (which, for DRAM devices, whose design is fiercely cost-driven [120, 121], is proportional to cost). Techniques range from the relatively low-cost (and widely employed) parity, SEC-DED (single error correction, double error detection), Chipkill [35], and DEC-TED (double error correction, triple error detection), all of which use different *error-correcting codes* (ECC) to detect and correct a small number of bits or chip errors, to the more expensive RAIM [111] and Mirroring [53] techniques that replicate some (or all) of memory to tolerate the failure of an entire DRAM dual in-line memory module (DIMM). The additional cost of memory with high error-tolerance can be significant (e.g., 12.5% of the total memory capacity for SEC-DED and Chipkill, and as high as 125% for Mirroring).

Table 1: Memory error detection and correction techniques. “ $X/Y\ Z$ ” means a technique can detect/correct X out of every Y failures of Z . n represents the parity of any odd number of bits between 1 and 63. Adapted from [104].

Technique	Error Detection (Correction)	Added Capacity	Added Logic
Parity	$n/64$ bits (None)	1.6%	Low
SEC-DED	$2/64$ bits ($1/64$ bits)	12.5%	Low
DEC-TED	$3/64$ bits ($2/64$ bits)	23.4%	Low
Chipkill [35]	$2/8$ chips ($1/8$ chips)	12.5%	High
RAIM [111]	$1/5$ modules ($1/5$ modules)	40.6%	High
Mirroring [53]	$2/8$ chips ($1/2$ modules)	125.0%	Low

Yet even with well-tested and error-tolerant memory devices, recent studies from the field have observed a rising rate of memory error occurrences [55, 72, 116, 120, 123, 141, 146]. This trend presents an increasing challenge for ensuring high performance and high reliability in future systems, as memory errors can be detrimental to both. In terms of performance, existing error detection and correction techniques incur a slowdown on each memory access due to their additional circuitry [55, 92] and up to an additional 10% slowdown due to techniques that operate DRAM at a slower speed to reduce the chances of random bit flips due to electrical interference in higher-density devices that pack more and more cells per square nanometer [148]. In addition, whenever an error is detected or corrected on modern hardware, the processor raises an interrupt that must be serviced by the system firmware (e.g., BIOS), incurring up to 100 μ s latency—roughly $2000\times$ the latency of a typical 50 ns memory access latency [58]—leading to unpredictable slowdowns and sometimes even system hangs [116].

In terms of reliability, memory errors can cause an application to slow down, hang, crash, or produce incorrect results [40]. Software-level techniques such as the retirement of regions of memory with errors [55, 76, 116, 118, 150] have been proposed to reduce the occurrence of memory error correction events and prevent correctable errors from turning into uncorrectable errors over time. Hardware-level techniques, such as those listed in Table 1, are used to detect and correct errors without software intervention (but *with*

additional hardware cost). All of these techniques are applied *homogeneously* to memory systems in a *one-size-fits-all* manner.

Our goal in our DSN 2014 paper [104] is to (1) understand how tolerant different data-intensive applications and different memory regions of each application are to memory errors, and (2) design a new memory system organization that matches hardware reliability to the error tolerance of the application and the memory region in order to reduce system cost. The **main idea** of our approach is to classify applications and memory regions based on their memory error tolerance, and map applications and memory regions to *heterogeneous-reliability* memory (HRM) system designs managed cooperatively between hardware and software to reduce system cost. We make the following **contributions**:

1. A new **methodology** to quantify the tolerance of applications and their memory regions to memory errors. Our approach measures the effect of memory errors on application correctness and quantifies an application’s ability to mask or recover from memory errors.
2. A comprehensive **characterization** of the memory error tolerance of three data-intensive workloads: an interactive web search application [104, 138], an in-memory key-value store [34, 104], and a graph mining framework [103, 104]. We find that there exists an order of magnitude difference in memory error tolerance across these three applications. We also find that there exists an order of magnitude difference in memory error tolerance across different memory regions of each application.
3. An **exploration** of the design space of a family of new memory system organizations, called *heterogeneous-reliability memory*, which combines a heterogeneous mix of reliability techniques that leverage application and memory region error tolerance to reduce system cost. We show that an example use of our techniques reduces server hardware cost by 4.7%, while achieving 99.90% single server availability, based on a preliminary evaluation of an example HRM system.

2. Characterizing Memory Error Tolerance

We characterize three commonly-used data-intensive applications to quantify their tolerance to memory errors:

- **WebSearch** [138], an interactive web search application,
- **Memcached** [34], an in-memory key-value store, and
- **GraphLab** [103], a graph mining framework.

We run these three applications in *real production systems*, and sample hundreds to tens of thousands of unique memory addresses for each application.

2.1. Characterization Methodology

To understand how tolerant different data-intensive applications are to memory errors, our characterization consists of three components: (1) characterizing the outcomes of memory errors on an application based on how they propagate

through an application's code and data, (2) characterizing how safe or unsafe it is for memory errors to occur in different regions of an application's data, and (3) determining how amenable an application's data is to recovery in the event of an error. We describe the implementation of each component in detail in Sections III and IV of our DSN 2014 paper [104].

We characterize an application's vulnerability to a memory error based on its behavior after a memory error is introduced (we assume for the moment that no error detection or correction is being performed). Figure 1 shows a taxonomy of memory error outcomes. Our taxonomy is mutually exclusive (no two outcomes occur simultaneously) and exhaustive (it captures all possible outcomes). At a high level, a memory error may be either (1) masked by an overwrite, in which case it is never detected and causes no change in application behavior; or (2) consumed by the application. In the case that an error is consumed by the application, it may either (2.1) be masked by application logic, in which case it is never detected and causes no change in application behavior; (2.2) cause the application to generate an incorrect response; or (2.3) cause the application or system to crash.

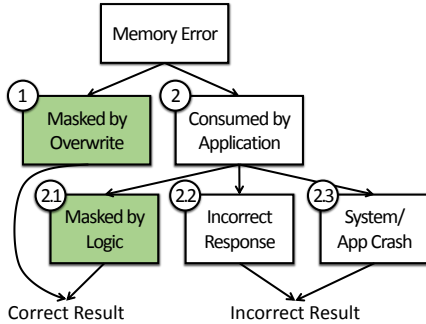


Figure 1: Memory error outcomes. Reproduced from [104].

When we refer to the tolerance of an application to memory errors, we mean the likelihood of an error occurring in some data results in outcomes (1) or (2.1). Conversely, when we refer to the vulnerability of an application to memory errors, we mean the likelihood of an error occurring in some data results in outcomes (2.2) or (2.3).

We have three design goals when implementing our methodology for quantifying application memory error tolerance. First, due to the sporadic and inconsistent nature of memory errors in the field [65, 72, 100, 116, 135, 141, 145, 146, 147], we want to design a framework that emulates the occurrence of a memory error in an application's data in a **controlled** manner. Second, we want an **efficient** way to measure how an application accesses its data. Third, we want our framework to be easily **adaptable** to other workloads or system configurations.

Figure 2 shows a flow diagram illustrating the five steps involved in our error emulation framework. We assume that the application under examination has already been run outside of the framework and its expected output without any memory errors has been recorded. The framework proceeds

as follows. (1) We start the application under the error injection framework. Our memory error emulation framework is described in Section IV of our DSN 2014 paper [104]. (2) We use software debuggers¹ to inject the desired number and types of memory errors. (3) We initiate the connection of a client and start executing the desired workload. (4) Throughout the course of the application's execution, we check to see if the machine has crashed; if it has, we log this outcome and proceed to step (1) to begin testing once again. (5) If the application finishes its workload, we check to see if its output matches the expected results; if the output does not match the expected results, we log this outcome and proceed to step (1) to test again. Each run injects a particular pattern of errors into the application. We can run this framework as many times as needed to test an application with different patterns of injected errors.

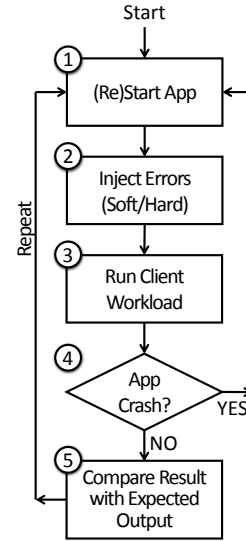


Figure 2: Memory error emulation framework. Reproduced from [104].

There are two main types of memory errors: (1) *soft* or *transient* errors and (2) *hard* or *recurring* errors.² Soft memory errors occur at random due to charged particle emissions from chip packaging or the atmosphere [110]. Hard memory errors may occur from physical device defects or wear-out [55, 141, 146], and are influenced by environmental factors such as humidity, temperature, and utilization [141, 144, 147]. Hard errors typically affect multiple bits (for example, large memory regions and entire DRAM chips have been shown to fail [55, 146, 147]). Our characterization covers single-bit soft and hard errors. For a detailed background on DRAM, we refer the reader to prior works [24, 25, 26, 27, 51, 52, 66, 71, 72, 73, 74, 75, 83, 84, 85, 86, 87, 99, 100, 131, 142, 143].

¹WinDBG [119] in Windows and GDB [42] in Linux.

²Recent studies [62, 64, 65, 72, 100, 135] examined the effects of *intermittent* and *access-pattern dependent* errors, which are increasingly common as DRAM technology scales down to smaller technology nodes [120].

2.2. Key Findings

We summarize two of the most important findings from our characterization below. We briefly list four other findings in Section 2.3, and describe all six of our findings in detail in Section V-B of our DSN 2014 paper [104].

Finding 1: Error Tolerance Varies Across Applications. Figure 3(a) plots the probability of each of the evaluated three applications crashing due to the occurrence of single-bit soft or hard errors in their memory (we call this *application-level memory error vulnerability*). For cases where the application does not crash, Figure 3(b) plots the rate of incorrect results per billion application queries under the same conditions. We draw two key observations from these results.

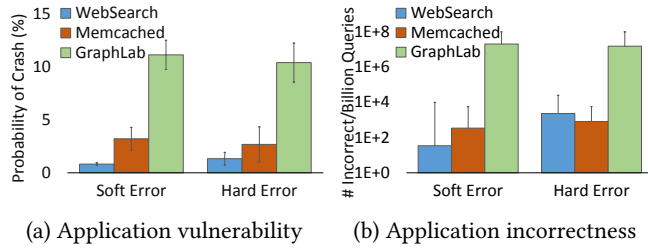
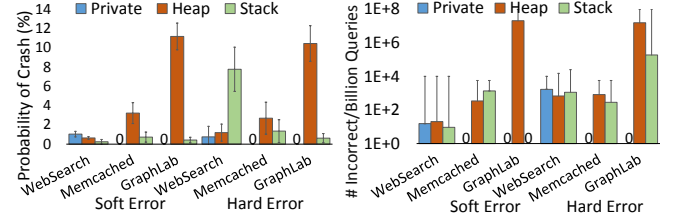


Figure 3: Inter-application variations in vulnerability to single-bit soft and hard memory errors for the three applications in terms of (a) probability of crash and (b) frequency of incorrect results. Reproduced from [104].

First, there exists a significant variance in vulnerability among the three applications both in terms of crash probability and in terms of incorrect result rate, which varies by up to six orders of magnitude. Second, these characteristics may differ depending on whether errors are soft or hard (for example, the number of incorrect results for WebSearch differs by over two orders of magnitude between soft and hard errors, with hard errors being more problematic). We therefore conclude that *memory reliability techniques that treat all applications similarly are inefficient because there exists significant variation in error tolerance among applications*.

Finding 2: Error Tolerance Varies Within an Application. Figure 4(a) plots the probability of each of the three applications crashing due to the occurrence of single-bit soft or hard errors in *different regions* of their memory address space. Figure 4(b) plots the rate of incorrect results per billion queries under the same conditions, for cases where a crash did not occur.

We make two observations from Figure 4. First, for some memory regions, the probability of an error leading to a crash is much lower than for others (for example, in WebSearch, the probability of a hard error leading to a crash in the *heap* or *private* memory regions is much lower than in the *stack* memory region). Second, even in the presence of memory errors, some regions of some applications are still able to tolerate memory errors (perhaps at reduced correctness). This may be acceptable for applications such as WebSearch that aggregate results from several servers before presenting them to the



(a) Memory region vulnerability (b) Memory region incorrectness

Figure 4: Memory region variations in vulnerability to single-bit soft and hard memory errors for the applications in terms of (a) probability of crash and (b) frequency of incorrect results. Reproduced from [104].

user, in which case the likelihood of the user being exposed to an error is much lower than the reported probabilities. We therefore conclude that *memory reliability techniques that treat all memory regions within an application similarly are inefficient because there exists significant variance in the error tolerance among different memory regions*.

2.3. Other Findings

In Section V-B of our DSN 2014 paper [104], we discuss four other findings that we make based on our characterization data. These findings focus on the memory error tolerance of WebSearch, which we find to be representative of the behavior of all three of our characterized applications. In particular, we find that:

- More severe failures (i.e., failures that lead to system down-times) due to memory errors tend to crash the application or system quickly, while less severe failures tend to generate incorrect results periodically.
- Some memory regions are safer than others. This indicates that either an application’s access pattern or computational operations on different memory regions can be the dominant factor to mask a majority of memory errors.
- More severe errors mainly *decrease* correctness, as opposed to increase an application’s probability of crashing.
- Data recoverability varies across memory regions. For data-intensive applications like WebSearch, software-only memory error tolerance techniques are a promising direction for enabling reliable system designs.

3. Heterogeneous-Reliability Memory

Based on the findings from our experimental characterization, we propose **heterogeneous-reliability memory (HRM)**, a software/hardware cooperative framework that employs different levels of memory reliability within a single main memory subsystem to optimize datacenter cost based on the memory error tolerance level of applications and their memory regions. We examine three dimensions, and their benefits and trade-offs in the design space, for systems with heterogeneous reliability memory: (1) hardware techniques to detect and correct errors, (2) software responses to errors, and (3) the granularity at which different techniques are used.

Table 2 lists the techniques we considered in each of the dimensions along with their potential benefits and trade-offs.

Using WebSearch as an example application, we evaluate and compare five example design points (three non-HRM systems, and two HRM systems):

- **Typical Server (non-HRM):** A baseline configuration resembling a typical server deployed in a modern datacenter. All memory is homogeneously protected using SEC-DED ECC.
- **Consumer PC (non-HRM):** Consumer PCs typically have no hardware protection against memory errors, reducing both their cost and reliability.
- **Detect&Recover (HRM):** Based on our observation that some memory regions are safer than others, we consider an HRM system design that, for the *private* region, uses parity in hardware to detect errors and responds by correcting them with a clean copy of data from disk in software (Par+R, parity and recovery), and uses neither error detection nor correction for the rest of its data.
- **Less-Tested (L; non-HRM):** Testing increases both the cost and average reliability of memory devices [120, 121, 131]. This system examines the implications of using less-thoroughly-tested memory throughout the *entire* memory system.
- **Detect&Recover/L (HRM):** This system evaluates the Detect&Recover design with less-tested memory. ECC is used in the *private* region and Par+R in the *heap* to compensate for the reduced reliability of the less-tested memory.

Section VI-A of our DSN 2014 paper [104] discusses (1) the metrics we use to evaluate the benefits and costs of the designs, and (2) the memory error model we use to examine the effectiveness of the five designs. We refer the reader to Section VI-A in [104] for detail and a full understanding.

Our evaluation illustrates the inefficiencies of traditional homogeneous approaches to memory system reliability, as

well as the benefits of heterogeneous-reliability memory system designs. Figure 5 shows the cost savings and single server availability for our five evaluated design points. We observe from the figure that the two highlighted example HRM design points (in orange color), which leverage our *heterogeneous-reliability* memory system design, both can achieve our target single server availability of 99.90% while reducing server hardware cost by 2.9% and 4.7% respectively. We therefore conclude that *heterogeneous-reliability memory system designs can enable systems to achieve both high cost savings and high single server availability/reliability at the same time.*

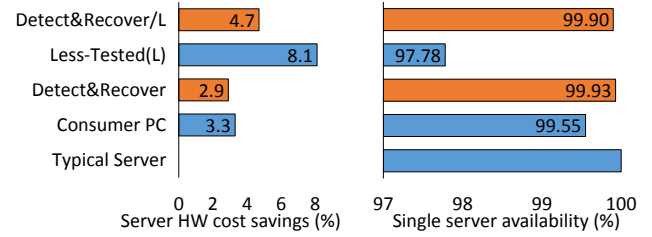


Figure 5: Comparison of server hardware cost savings and single server availability for the five design points. Results extracted from [104]. Orange bars indicate HRM designs.

Section VI of our DSN 2014 paper [104] contains a detailed analysis of HRM, including (1) memory cost savings (Section VI-B of [104]), (2) the expected crash and incorrect query frequency for each configuration (Section VI-B of [104]), (3) the maximum number of tolerable errors per month for each application to achieve a reliability target (Section VI-B of [104]), and (4) a discussion of hardware/software support for and feasibility of HRM (Section VI-C of [104]). We summarize the key empirical findings here:

- Our two example HRM designs, Detect&Recover and Detect&Recover/L, reduce memory costs by 9.7% and 15.5%,

Table 2: Heterogeneous reliability design dimensions, example techniques, and their potential benefits and trade-offs. Adapted from [104].

Design dimension	Technique	Benefits	Trade-offs
Example hardware techniques	No detection/correction	No associated overheads (low cost)	Unpredictable crashes and silent data corruption
	Parity	Relatively low cost with detection capability	No hardware correction capability
	SEC-DED/DEC-TED	Tolerate common single-/double-bit errors	Increased cost and memory access latency
	Chipkill [35]	Tolerate single-DRAM-chip errors	Increased cost and memory access latency
	Mirroring [53]	Tolerate memory module failure	100% capacity overhead
Example software responses	Less-Tested DRAM	Saved testing cost during manufacturing	Increased error rates
	Consume errors in application	Simple, no performance overhead	Unpredictable crashes and data corruption
	Automatically restart application	Can prevent unpredictable application behavior	May make little progress if error is frequent
	Retire memory pages	Low overhead, effective for repeating errors	Reduces memory space (usually very little)
	Conditionally consume errors	Flexible, software vulnerability-aware	Memory management overhead to make decision
Usage granularity	Software correction	Tolerates detectable memory errors	Usually has performance overheads
	Physical machine	Simple, uniform usage across memory space	Costly depending on technique used
	Virtual machine	More fine-grained, flexible management	Host OS is still vulnerable to memory errors
	Application	Manageable by the OS	Does not leverage different region tolerance
	Memory region	Manageable by the OS	Does not leverage different page tolerance
	Memory page	Manageable by the OS	Does not leverage different data object tolerance
	Cache line	Most fine-grained management	Large management overhead; software changes

respectively, compared to the cost of the Typical Server system, which does not use HRM.

- The two example HRM designs limit the number of crashes to 3 and 4 per server per month, respectively, and limit the incorrect query frequency to 9 and 12 per million queries, respectively.
- Without *any* error detection/correction, two out of our three evaluated applications (WebSearch and Memcached) are able to achieve 99.00% single server availability.

We therefore conclude that heterogeneous-reliability memory system designs can enable systems to achieve both high cost savings and high single server availability/reliability at the same time. We believe that there is significant opportunity in many data-intensive applications for reducing server hardware cost while achieving high single server availability/reliability using our heterogeneous-reliability design methodology.

4. Related Work

To our knowledge, our DSN 2014 paper [104] is the first to (1) perform a comprehensive analysis of memory error vulnerability for *data-intensive datacenter applications across a range of different memory error types*; (2) propose the idea of *heterogeneous reliability memory*, which consists of multiple memory types with different levels of reliability and error handling mechanisms; and (3) evaluate the cost-effectiveness of different *heterogeneous-reliability memory organizations* with hardware/software cooperation. We discuss related research in memory error vulnerability and DRAM architecture below, categorizing the works into six broad classes: (1) memory errors in datacenters, (2) characterizing application error tolerance, (3) hardware-based memory reliability techniques, (4) software-based memory reliability techniques, (5) exploiting application error tolerance, and (6) heterogeneous (hybrid) memory architectures.

Studies of Memory Errors. Various works [92, 116, 141, 145, 146, 147] have conducted studies of DRAM error rates that are deployed in production datacenters, studying failures across a large sample size. These works note that memory errors occur frequently in datacenters, and are induced by a number of error sources. In particular, one of these studies empirically demonstrates the increased memory errors and increased memory cost to tolerate these errors in large-scale datacenters [116]. A recent work [48] examines how various hardware and software techniques to detect and mitigate errors introduce significant performance degradation in production datacenters. This work shows that for WebSearch, software error handling techniques can induce a performance overhead of $3746\times$ [48]. These studies motivate the need for a low-overhead, cost-effective approach to memory reliability, and motivate us to further explore hardware–software cooperative techniques such as HRM.

There are several studies that characterize various sources of errors in DRAM at a fine granularity. Many of these

works observe how specific factors affect DRAM errors, analyzing the impact of temperature [37, 86] and hard errors [55]. A large number of works study errors through controlled experiments, usually using FPGA-based DRAM testing infrastructures like SoftMC [51], to investigate errors due to retention time [51, 62, 63, 64, 65, 99, 100, 131, 135], disturbance from neighboring DRAM cells [60, 70, 72, 120], latency variation across/within DRAM chips [21, 23, 25, 82, 83, 86], and supply voltage [23, 27]. None of these works study memory errors in a system with heterogeneous-reliability memory.

Classifying Application Error Tolerance. Error injection techniques based on hardware watchpoints [92, 112], binary instrumentation [89], and architectural simulation [93] have been used to investigate the impact of memory errors on application behavior, including execution times, application/system crashes, and output correctness. These works study a range of applications including SPEC CPU benchmarks, web servers, databases, and scientific applications. In general, these works conclude that not all memory errors cause application/system crashes and many memory errors can be tolerated with minimal difference in the application outputs. We generalize this observation to data-intensive applications, and leverage it to reduce datacenter TCO. Recent work [149] develops a Markov-chain model for the error tolerance of HPC applications. Approximate computing techniques [8, 39, 57], where the precision of program output can be relaxed to achieve better performance or energy efficiency, offer further opportunities for leveraging the error-tolerance of application data, though these typically require very careful changes to the program source code.

Hardware-Based Memory Reliability Techniques. There are various ECC techniques for memory, and we list the most dominant ones in Table 1. Using eight bits, SEC-DED can correct a single bit flip and detect up to two bit flips out of every 64 bits. DEC-TED is a generalization of SEC-DED that uses fourteen bits to correct two and detect three flipped bits out of every 64 bits. Chipkill [35] improves reliability by interleaving error detection and correction data among multiple DRAM chips. RAIM [111] is able to tolerate entire DIMMs failing by storing detection and correction data across multiple DIMMs. Virtualized ECC [155] maps ECC to software-visible locations in memory so that software can decide what ECC protection to use. While Virtualized ECC can help reduce the DRAM hardware cost of memory reliability, it requires modification to the processor’s memory management unit and cache(s).

Recent works propose new hardware-based techniques to tolerate soft and hard memory errors efficiently. We break these down into four categories: (1) *Tolerating soft errors*: BambooECC [67] proposes a new single-tier ECC family that enables adaptive graceful downgrade of ECC capabilities. CleanECC [45] provides both high memory reliability and flexible memory access granularity by using fine-grained error detection and coarse-grained error correction. XED [126]

uses in-DRAM ECC to reduce the overhead of double Chip-kill. (2) *Tolerating hard errors*: ArchShield [124] proposes an architectural framework to identify and tolerate hard errors caused by DRAM cell failures. Citadel [125] proposes to tolerate large-granularity failures, such as row/bank failures, by replacing them with spares. Other works propose to identify and mitigate potentially recurring memory errors by page offlining [116], online testing [65, 131], and multi-rate refresh [99, 135]. (3) *Reducing memory cost*: FrugalECC [68] proposes a new flexible granularity compression to reduce the redundancy and energy consumption of ECC. Morphable ECC [29] proposes to reduce DRAM refresh overhead by reducing ECC strength to 6-bit ECC when the DRAM is in idle mode. (4) *End-to-end memory error protection*: AIECC [69] provides end-to-end protection for clock, control, command, and address (CCCA) signals in addition to data signals.

Software-Based Memory Reliability Techniques. Previous works (e.g., [55, 116, 140, 150]) show that the OS retiring memory pages after a certain number of errors can eliminate up to 96.8% of detected memory errors. While these techniques improve system reliability, they still require costly ECC hardware for detecting and identifying memory pages with errors. Other works attempt to reduce the impact of memory errors on system reliability by writing more reliable software [7], modifying the OS memory allocator [132], or using a compiler to generate a more error-tolerant version of the program [5, 22]. Other algorithmic solutions (e.g., memory bounds checks [88], watchdog timers [88], and checkpoint recovery [30, 31, 32, 90, 91, 95, 96, 97, 153, 154]) can also be used to improve resilience to memory errors.

Li et al. [98] propose to deploy software-based ECC in an in-memory key-value store, and show that it incurs low performance overhead. Recent works [149, 161] improve upon traditional RAIM-3 and use selective replication to reduce unnecessary memory redundancy. SDECC [46, 47] proposes to use strong error detection in the hardware, while tolerating hard memory errors and recovering from soft errors in the software.

Exploiting Application Error Tolerance. Flikker [102] proposes a technique to trade off DRAM reliability for energy savings. It relies on the programmer to separate application data into vulnerable or tolerant data. Less reliable mobile DIMMs have been proposed [109, 156] as a replacement for ECC DIMMs in servers to improve energy efficiency. Recent work [128] shows that RAMCloud can recover 35 GB of data from a failed server in 1.6 seconds using a log-structure storage.

Heterogeneous (Hybrid) Memory Architectures. Various recent works (e.g., [1, 6, 28, 36, 44, 94, 101, 113, 114, 133, 134, 136, 137, 157, 158, 160]) explore the use of heterogeneous memory architectures, consisting of multiple different types of memories. These works are mainly concerned with either mitigating the overheads of emerging memory technologies or improving performance and power efficiency. They do not

investigate the use of multiple devices with different *error correction* capabilities. CREAM [107] and Odd-ECC [108] develop low-cost techniques to provide flexible provisioning of memory error correction capabilities. Recent works [3, 152] apply our heterogeneous reliability idea to processor caches to achieve better cost-reliability trade-offs.

5. Significance and Long-Term Impact

We believe that our DSN 2014 paper [104] will have long-term impact for three major reasons. First, it emphasizes and aims to solve the increasing cost of ensuring memory reliability as the error rates of memory devices continue to grow, which is a major trend as memory technology scales to smaller technology nodes [120, 121]. Second, it tackles memory system cost in datacenters, which is a problem that we expect will be increasingly important in the future. Third, it proposes a novel framework that uses hardware-software co-design to improve memory system reliability as well as cost, thereby hopefully inspiring future works to exploit software characteristics to improve system reliability and reduce system cost (and other important metrics).

Increasing Memory Error Rate. As DRAM scales to smaller process technology nodes, the reliability of DRAM continues to degrade [55, 61, 116, 120, 121, 122, 123, 141, 145, 146, 147]. For example, recent works 1) show the existence of disturbance errors in commodity DRAM chips operating in the field [72, 120]; 2) experimentally demonstrate the increasing importance of retention-related failures in modern DRAM devices [51, 62, 63, 64, 65, 99, 100, 131, 131, 135]; 3) examine the trade-off between DRAM reliability and latency [21, 23, 25, 27, 51, 66, 82, 83, 86]; and 4) advocate, including in a paper co-written by the Samsung and Intel memory design teams [61], for the use of in-DRAM error correcting codes to overcome the reliability challenges [61, 135]. As a result of decreasing DRAM reliability, maintaining the effective error rate at the levels we have today can (1) increase DRAM cost due to decreased yield, expensive quality assurance tests, and/or extra capacity for storing stronger error-correcting codes; or (2) reduce DRAM performance due to frequent error correction and logging. All of these solutions might make DRAM technology scaling more difficult and less appealing [120, 121, 122]. Our paper proposes a solution that enables the use of DRAM with *higher* error rates while still achieving reasonable application reliability, which can enable much more efficient scaling of DRAM to smaller technology nodes in the future.

Other memory technologies such as NAND flash memory [9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 43, 105, 106, 115], phase-change memory (PCM) [79, 80, 81, 114, 117, 136] and STT-MRAM [78, 114] also show a similar decreasing trend in their reliability with process technology scaling and the advent of multi-level cell (MLC) technology [121]. For example, like DRAM, NAND flash memory suffers from retention errors [9, 10, 11, 13, 14, 15], cell-to-cell program in-

terference errors [9, 10, 11, 14, 17, 19], and read disturb errors [9, 10, 11, 14, 20]. Additionally, NAND flash memory suffers from program/erase cycling errors [14, 18], and programming errors [12, 105, 130]. PCM suffers from endurance issues [79, 81, 136] and resistance drift [56]. HRM can be applied to these memory technologies with slight modifications to enable reliable high-density non-volatile devices in the future.

Increasing Datacenter Cost. Recent studies have shown that capital costs can account for the majority (e.g., around 57% in [4]) of datacenter TCO (total cost of ownership). As part of the cost of a server, the cost of the memory is comparable to that of the processors [77], and is likely to exceed processor cost and become the dominant cost for servers running data-intensive applications such as web search and social media services [34, 103, 138]. As future datacenters grow in scale, datacenter TCO will become an increasingly important factor in system design. Our paper demonstrates a way of optimizing datacenter TCO by reducing the cost of the memory system. The cost savings can be significant due to the increasing scale of such datacenters [50], making our proposed technique hopefully more important in the future.

Hardware-Software Co-Design. Our solution, heterogeneous-reliability memory, utilizes hardware-software cooperative design to reduce system cost. Our DSN 2014 paper [104] demonstrates the benefits of exploiting application characteristics to improve overall system design. For example, it shows that a significant number of errors can be corrected in software by reloading a clean copy of the data from storage. This motivates us to rethink the placement of different functionalities (such as error detection and error correction) across different system components and across software versus hardware to improve the cost-reliability trade-off.

Our DSN 2014 paper [104] has started a community discussion [50] on the feasibility of solving the problem of memory reliability by exploiting application memory error tolerance in the future, inspiring reporters to ask the question: “How good does memory need to be?” We hope that our characterization results and mechanisms will hopefully continue to inspire future works that can provide efficient and extensive characterization/estimation of application-level memory error tolerance [41], which can make our proposed technique applicable to a broader set of applications.

Two example works that build on ours include Odd-ECC [108] and CREAM [107]. Odd-ECC provides a mechanism to enable different levels of fault tolerance for the data stored in a commodity DRAM module. Odd-ECC maps the ECC bits to a memory address aligned with the data so that the memory controller can access both the data and the ECC bits efficiently. CREAM provides a mechanism to dynamically adjust the tradeoff between memory capacity/bandwidth used for ECC bits and fault tolerance within an ECC DRAM module. CREAM proposes several data layouts that reduce page

faults and improve memory performance significantly when strong fault tolerance is not needed.

6. Conclusion

In our DSN 2014 paper [104], we develop a new methodology to quantify the tolerance of applications to memory errors. Using this methodology, we perform a case study of three new data-intensive workloads that show, among other new insights, that there exists a diverse spectrum of memory error tolerance both within and across these applications. Based on this observation, we introduce the idea of heterogeneous-reliability memory (HRM), which combines multiple different memories that have different reliability characteristics and error correction capabilities. We propose new hardware/software heterogeneous-reliability memory system designs, and evaluate them to show that (1) the one-size-fits-all approach to reliability in modern servers is inefficient in terms of cost, and (2) heterogeneous-reliability systems can achieve the benefits of both low cost and high single server availability/reliability. We hope that our techniques can enable the use of lower-cost memory devices to reduce the server hardware cost of datacenters, and that our analyses will spur future research on heterogeneous-reliability memory systems. As DRAM technology scales into small feature sizes and becomes less reliable and memory cost becomes more important in datacenters in the future, we hope that our findings and ideas will inspire more research to improve the cost-reliability trade-off in memory systems. We believe different HRM designs can be employed to optimize other key trade-offs and target metrics (e.g., performance vs. energy consumption) in modern systems. Our DSN 2014 paper just scratches the surface of a large amount of research and design space to be explored.

Acknowledgments

We thank Saugata Ghose for his dedicated effort in the preparation of this article. We thank the anonymous reviewers and the members of SAFARI research group for feedback. We acknowledge the support of Microsoft and Samsung. This research was partially supported by the Intel Science and Technology Center for Cloud Computing and the NSF (grants 0953246, 1065112, and 1212962).

References

- [1] N. Agarwal and T. F. Wenisch, “Thermostat: Application-Transparent Page Management for Two-Tiered Main Memory,” in *ASPLOS*, 2017.
- [2] Z. Al-Ars, “DRAM Fault Analysis and Test Generation,” Ph.D. dissertation, Delft, 2005.
- [3] S. Arslan, H. R. Topcuoglu, M. T. Kandemir, and O. Tosun, “Performance and Energy Efficient Asymmetrically Reliable Caches for Multicore Architectures,” in *IPDPSW*, 2015.
- [4] L. A. Barroso et al., *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool Publishers, 2009.
- [5] A. Benso et al., “A C/C++ Source-to-Source Compiler for Dependable Applications,” in *DSN*, 2000.
- [6] S. Bock, B. R. Childers, R. Melhem, and D. Mossé, “Concurrent Migration of Multiple Pages in Software-Managed Hybrid Main Memory,” in *ICCD*, 2016.
- [7] C. Borchert et al., “Generative Software-Based Memory Error Detection and Correction for Operating System Data Structures,” in *DSN*, 2013.

- [8] J. Bornholt *et al.*, "Uncertain<T>: A First-order Type for Uncertain Data," in *ASPLOS*, 2014.
- [9] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, and O. Mutlu, "Error Characterization, Mitigation, and Recovery in Flash-Memory-Based Solid-State Drives," *Proc. IEEE*, Sep. 2017.
- [10] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, and O. Mutlu, "Error Characterization, Mitigation, and Recovery in Flash Memory Based Solid-State Drives," arXiv:1706.08642 [cs.AR], 2017.
- [11] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, and O. Mutlu, "Errors in Flash-Memory-Based Solid-State Drives: Analysis, Mitigation, and Recovery," arXiv:1711.11427 [cs.AR], 2017.
- [12] Y. Cai, S. Ghose, Y. Luo, K. Mai, O. Mutlu, and E. F. Haratsch, "Vulnerabilities in MLC NAND Flash Memory Programming: Experimental Analysis, Exploits, and Mitigation Techniques," in *HPCA*, 2017.
- [13] Y. Cai, Y. Luo, E. F. Haratsch, K. Mai, and O. Mutlu, "Data Retention in MLC NAND Flash Memory: Characterization, Optimization, and Recovery," in *HPCA*, 2015.
- [14] Y. Cai *et al.*, "Error Patterns in MLC NAND Flash Memory: Measurement, Characterization, and Analysis," in *DATE*, 2012.
- [15] Y. Cai *et al.*, "Flash Correct-and-Restore: Retention-Aware Error Management for Increased Flash Memory Lifetime," in *ICCD*, 2012.
- [16] Y. Cai *et al.*, "Error Analysis and Retention-Aware Error Management for NAND Flash Memory," *ITJ*, 2013.
- [17] Y. Cai *et al.*, "Program Interference in MLC NAND Flash Memory: Characterization, Modeling, and Mitigation," in *ICCD*, 2013.
- [18] Y. Cai *et al.*, "Threshold Voltage Distribution in MLC NAND Flash Memory: Characterization, Analysis, and Modeling," in *DATE*, 2013.
- [19] Y. Cai *et al.*, "Neighbor-Cell Assisted Error Correction for MLC NAND Flash Memories," in *SIGMETRICS*, 2014.
- [20] Y. Cai, Y. Luo, S. Ghose, and O. Mutlu, "Read Disturb Errors in MLC NAND Flash Memory: Characterization, Mitigation, and Recovery," in *DSN*, 2015.
- [21] K. Chandrasekar, S. Goossens, C. Weis, M. Koedam, B. Akesson, N. Wehn, and K. Goossens, "Exploiting Expendable Process-Margins in DRAMs for Run-Time Performance Optimization," in *DATE*, 2014.
- [22] J. Chang *et al.*, "Automatic Instruction-Level Software-Only Recovery," in *DSN*, 2006.
- [23] K. K. Chang, "Understanding and Improving the Latency of DRAM-Based Memory Systems," Ph.D. dissertation, Carnegie Mellon Univ., 2017.
- [24] K. K. Chang, D. Lee, Z. Chishti, A. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu, "Improving DRAM Performance by Parallelizing Refreshes With Accesses," in *HPCA*, 2014.
- [25] K. K. Chang, A. Kashyap, H. Hassan, S. Ghose, K. Hsieh, D. Lee, T. Li, G. Pekhimenko, S. Khan, and O. Mutlu, "Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization," in *SIGMETRICS*, 2016.
- [26] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu, "Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM," in *HPCA*, 2016.
- [27] K. K. Chang, A. G. Yaglikci, A. Agrawal, N. Chatterjee, S. Ghose, A. Kashyap, H. Hassan, D. Lee, M. O'Connor, and O. Mutlu, "Understanding Reduced-Voltage Operation in Modern DRAM Devices: Experimental Characterization, Analysis, and Mechanisms," in *SIGMETRICS*, 2017.
- [28] N. Chatterjee *et al.*, "Leveraging Heterogeneity in DRAM Main Memories to Accelerate Critical Word Access," in *MICRO*, 2012.
- [29] C. Chou, P. Nair, and M. K. Qureshi, "Reducing Refresh Power in Mobile Devices with Morphable ECC," in *DSN*, 2015.
- [30] K. Constantinides, O. Mutlu, and T. Austin, "Online Design Bug Detection: RTL Analysis, Flexible Mechanisms, and Evaluation," in *MICRO*, 2008.
- [31] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco, "Software-Based Online Detection of Hardware Defects Mechanisms, Architectural Support, and Evaluation," in *MICRO*, 2007.
- [32] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco, "A Flexible Software-Based Framework for Online Detection of Hardware Defects," *TC*, 2009.
- [33] CST, Inc., "Memory Test Background," <http://tinyurl.com/m7c3wf7>, 2000.
- [34] Danga Interactive, "Memcached," <http://memcached.org/>.
- [35] T. J. Dell, "A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory," *IBM Microelectronics Division*, 1997.
- [36] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan, "Data Tiering in Heterogeneous Memory Systems," in *EuroSys*, 2016.
- [37] N. El-Sayed, I. A. Stefanovici, G. Amvrosiadis, A. A. Hwang, and B. Schroeder, "Temperature Management in Data Centers: Why Some (Might) Like It Hot," in *SIGMETRICS*, 2012.
- [38] E. M. Elnozahy *et al.*, "Energy-Efficient Server Clusters," in *PACS*, 2003.
- [39] H. Esmailzadeh *et al.*, "Architecture Support for Disciplined Approximate Programming," in *ASPLOS*, 2012.
- [40] D. Fiala *et al.*, "Detection and Correction of Silent Data Corruption for Large-scale High-performance Computing," in *SC*, 2012.
- [41] N. Fourtris *et al.*, "Versatile Architecture-level Fault Injection Framework for Reliability Evaluation: A first Report," in *IOLTS*, 2014.
- [42] Free Software Foundation, Inc., "GDB: The GNU Project Debugger," <http://www.sourceware.org/gdb/>.
- [43] A. Fukami, S. Ghose, Y. Luo, Y. Cai, and O. Mutlu, "Improving the Reliability of Chip-Off Forensic Analysis of NAND Flash Memory Devices," *Digital Investigation*, Mar 2017.
- [44] K. Gai, M. Qiu, H. Zhao, and L. Qiu, "Smart Energy-Aware Data Allocation for Heterogeneous Memory," in *HPCC*, 2016.
- [45] S.-L. Gong, M. Rhu, J. Kim, J. Chung, and M. Erez, "Clean-ECC: High reliability ECC for Adaptive Granularity Memory System," in *MICRO*, 2015.
- [46] M. Gottscho, I. Alam, C. Schoeny, L. Dolecek, and P. Gupta, "Low-Cost Memory Fault Tolerance for IoT Devices," *CASES/TECS*, 2017.
- [47] M. Gottscho, C. Schoeny, L. Dolecek, and P. Gupta, "Software-defined error-correcting codes," in *DSN Workshop*, 2016.
- [48] M. Gottscho, M. Shoaib, S. Govindan, B. Sharma, D. Wang, and P. Gupta, "Measuring the Impact of Memory Errors on Application Performance," *CAL*, 2017.
- [49] S. Grundberg *et al.*, "For Data Center, Google Goes for the Cold," <http://tinyurl.com/ml55nh5>, 2011.
- [50] R. Harris, "How good does memory need to be?" <http://www.zdnet.com/how-good-does-memory-need-to-be-7000031853/>, 2014.
- [51] H. Hassan, N. Vijaykumar, S. Khan, S. Ghose, K. Chang, G. Pekhimenko, D. Lee, O. Ergin, and O. Mutlu, "SoftMC: A Flexible and Practical Open-Source Infrastructure for Enabling Experimental DRAM Studies," in *HPCA*, 2017.
- [52] H. Hassan, G. Pekhimenko, N. Vijaykumar, V. Seshadri, D. Lee, O. Ergin, and O. Mutlu, "ChargeCache: Reducing DRAM Latency by Exploiting Row Access Locality," in *HPCA*, 2016.
- [53] D. Henderson *et al.*, "POWER7 System RAS," 2012.
- [54] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li, "An Analysis of Facebook Photo Caching," in *SOSP*, 2013.
- [55] A. A. Hwang, I. Stefanovici, and B. Schroeder, "Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design," in *ASPLOS*, 2012.
- [56] D. Ielmini *et al.*, "Physical Interpretation, Modeling and Impact on Phase Change Memory (PCM) Reliability of Resistance Drift Due to Chalcogenide Structural Relaxation," in *IEDM*, 2007.
- [57] Intel Corp., "iACT," <http://www.github.com/IntelLabs/iACT>.
- [58] JEDEC Solid State Technology Association, "JEDEC Standard: DDR3 SDRAM, JESD79-3C," 2008.
- [59] P. Jobin, "Cloud Computing Shifting to Cooler Climates," <http://tinyurl.com/mf1rlt>, 2012.
- [60] M. Jung, C. C. Rheinländer, C. Weis, and N. Wehn, "Reverse Engineering of DRAMs: Row Hammer with Crosshair," in *MEMSYS*, 2016.
- [61] U. Kang *et al.*, "Co-Architecting Controllers and DRAM to Enhance DRAM Process Scaling," in *The Memory Forum*, 2014.
- [62] S. Khan, D. Lee, and O. Mutlu, "PARBOR: An Efficient System-Level Technique to Detect Data-Dependent Failures in DRAM," in *DSN*, 2016.
- [63] S. Khan, C. Wilkerson, D. Lee, A. R. Alameldeen, and O. Mutlu, "A Case for Memory Content-Based Detection and Mitigation of Data-Dependent Failures in DRAM," *CAL*, 2016.
- [64] S. Khan, C. Wilkerson, Z. Wang, A. R. Alameldeen, D. Lee, and O. Mutlu, "Detecting and Mitigating Data-Dependent DRAM Failures by Exploiting Current Memory Content," in *MICRO*, 2017.
- [65] S. M. Khan *et al.*, "The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study," in *SIGMETRICS*, 2014.
- [66] J. S. Kim, M. Patel, H. Hassan, and O. Mutlu, "The DRAM Latency PUF: Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency-Reliability Tradeoff in Modern DRAM Devices," in *HPCA*, 2018.
- [67] J. Kim, M. Sullivan, and M. Erez, "Bamboo ECC: Strong, Safe, and Flexible Codes for Reliable Computer Memory," in *HPCA*, 2015.
- [68] J. Kim, M. Sullivan, S.-L. Gong, and M. Erez, "Frugal ECC: Efficient and Versatile Memory Error Protection Through Fine-Grained Compression," in *SC*, 2015.
- [69] J. Kim, M. Sullivan, S. Lym, and M. Erez, "All-Inclusive ECC: Thorough End-to-End Protection for Reliable Computer Memory," in *ISCA*, 2016.
- [70] Y. Kim, "Architectural Techniques to Enhance DRAM Scaling," Ph.D. dissertation, Carnegie Mellon Univ., 2015.
- [71] Y. Kim *et al.*, "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM," in *ISCA*, 2012.
- [72] Y. Kim *et al.*, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," in *ISCA*, 2014.
- [73] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers," in *HPCA*, 2010.
- [74] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *MICRO*, 2010.
- [75] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," *CAL*, 2015.
- [76] A. Kleen, "mcelog: Memory Error Handling in User Space," <http://www.halobates.de/lk10-mcelog.pdf>.
- [77] C. Kozyrakis *et al.*, "Server Engineering Insights for Large-Scale Online Services," *IEEE Micro*, 2010.

- [78] E. Kultursay *et al.*, "Evaluating STT-RAM As An Energy-Efficient Main Memory Alternative," in *ISPASS*, 2013.
- [79] B. C. Lee *et al.*, "Architecting Phase Change Memory as a Scalable DRAM Alternative," in *ISCA*, 2009.
- [80] B. C. Lee *et al.*, "Phase Change Memory Architecture and The Quest for Scalability," *CACM*, 2010.
- [81] B. C. Lee *et al.*, "Phase Change Technology and the Future of Main Memory," *IEEE Micro*, 2010.
- [82] D. Lee, "Reducing DRAM Energy at Low Cost by Exploiting Heterogeneity," Ph.D. dissertation, Carnegie Mellon Univ., 2016.
- [83] D. Lee, S. Khan, L. Subramanian, S. Ghose, R. Ausavarungnirun, G. Pekhimenko, V. Seshadri, and O. Mutlu, "Design-Induced Latency Variation in Modern DRAM Chips: Characterization, Analysis, and Latency Reduction Mechanisms," in *SIGMETRICS*, 2017.
- [84] D. Lee *et al.*, "Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture," in *HPCA*, 2013.
- [85] D. Lee, S. Ghose, G. Pekhimenko, S. Khan, and O. Mutlu, "Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost," *TACO*, 2016.
- [86] D. Lee, Y. Kim, G. Pekhimenko, S. Khan, V. Seshadri, K. Chang, and O. Mutlu, "Adaptive-Latency DRAM: Optimizing DRAM Timing for the Common-Case," in *HPCA*, 2015.
- [87] D. Lee, L. Subramanian, R. Ausavarungnirun, J. Choi, and O. Mutlu, "Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM," in *PACT*, 2015.
- [88] L. Leem *et al.*, "ERSA: Error Resilient System Architecture for Probabilistic Applications," in *DATE*, 2010.
- [89] D. Li *et al.*, "Classifying Soft Error Vulnerabilities in Extreme-Scale Scientific Applications Using a Binary Instrumentation Tool," in *SC*, 2012.
- [90] M.-L. Li *et al.*, "Trace-Based Microarchitecture-level Diagnosis of Permanent Hardware Faults," in *DSN*, 2008.
- [91] M.-L. Li *et al.*, "Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design," in *ASPLOS*, 2008.
- [92] X. Li *et al.*, "A Realistic Evaluation of Memory Hardware Errors and Software System Susceptibility," in *USENIX ATC*, 2010.
- [93] X. Li *et al.*, "Application-Level Correctness and Its Impact on Fault Tolerance," in *HPCA*, 2007.
- [94] Y. Li, S. Ghose, J. Choi, J. Sun, H. Wang, and O. Mutlu, "Utility-Based Hybrid Memory Management," in *CLUSTER*, 2017.
- [95] Y. Li, S. Makar, and S. Mitra, "CASP: Concurrent Autonomous Chip Self-Test Using Stored Test Patterns," in *DATE*, 2008.
- [96] Y. Li, O. Mutlu, D. S. Gardner, and S. Mitra, "Concurrent Autonomous Self-Test for Uncore Components in System-on-Chips," in *VTS*, 2010.
- [97] Y. Li, O. Mutlu, and S. Mitra, "Operating System Scheduling for Efficient Online Self-Test in Robust Systems," in *ICCAD*, 2009.
- [98] Y. Li, H. Wang, X. Zhao, H. Sun, and T. Zhang, "Applying Software-based Memory Error Correction for In-Memory Key-Value Store: Case Studies on Memcached and RAMCloud," in *MemSys*, 2016.
- [99] J. Liu *et al.*, "RAIDR: Retention-Aware Intelligent DRAM Refresh," in *ISCA*, 2012.
- [100] J. Liu *et al.*, "An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms," in *ISCA*, 2013.
- [101] L. Liu, H. Yang, Y. Li, M. Xie, L. Li, and C. Wu, "Memos: A Full Hierarchy Hybrid Memory Management Framework," in *ICCD*, 2016.
- [102] S. Liu *et al.*, "Flicker: Saving DRAM Refresh-Power Through Critical Data Partitioning," in *ASPLOS*, 2011.
- [103] Y. Low *et al.*, "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud," *PVLDB*, 2012.
- [104] Y. Luo *et al.*, "Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-Reliability Memory," in *DSN*, 2014.
- [105] Y. Luo, S. Ghose, Y. Cai, E. F. Haratsch, and O. Mutlu, "Enabling Accurate and Practical Online Flash Channel Modeling for Modern MLC NAND Flash Memory," *JSAC*, 2016.
- [106] Y. Luo, S. Ghose, Y. Cai, E. F. Haratsch, and O. Mutlu, "HeatWatch: Improving 3D NAND Flash Memory Device Reliability by Exploiting Self-Recovery and Temperature Awareness," in *HPCA*, 2018.
- [107] Y. Luo, S. Ghose, T. Li, S. Govindan, B. Sharma, B. Kelly, A. Boroumand, and O. Mutlu, "Using ECC DRAM to Adaptively Increase Memory Capacity," arXiv:1706.08870 [cs.AR], 2017.
- [108] A. Malek, E. Vasilakis, V. Papaefstathiou, P. Trancoso, and I. Sourdis, "Odd-ECC: On-Demand DRAM Error Correcting Codes," in *MemSys*, 2017.
- [109] K. T. Malladi *et al.*, "Towards Energy-proportional Datacenter Memory with Mobile DRAM," in *ISCA*, 2012.
- [110] T. C. May *et al.*, "Alpha-Particle-Induced Soft Errors in Dynamic Memories," *IEEE T-ED*, 1979.
- [111] P. J. Meaney *et al.*, "IBM zEnterprise Redundant Array of Independent Memory Subsystem," *IBM JRD*, 2012.
- [112] A. Messer *et al.*, "Susceptibility of Commodity Systems and Software to Memory Soft Errors," *IEEE TC*, 2004.
- [113] J. Meza *et al.*, "Enabling efficient and scalable hybrid memories using fine-granularity DRAM cache management," *CAL*, 2012.
- [114] J. Meza *et al.*, "A Case for Efficient Hardware/Software Cooperative Management of Storage and Memory," in *WEED*, 2013.
- [115] J. Meza *et al.*, "A Large Scale Study of Flash Errors in the Field," in *SIGMETRICS*, 2015.
- [116] J. Meza *et al.*, "Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field," in *DSN*, 2015.
- [117] J. Meza, J. Li, and O. Mutlu, "Evaluating Row Buffer Locality in Future Non-Volatile Main Memories," Carnegie Mellon Univ., SAFARI Research Group, Tech. Rep. TR-SAFARI-2012-002, 2012.
- [118] Microsoft Corp., "Predictive Failure Analysis (PFA)," <http://tinyurl.com/n34z657>.
- [119] Microsoft Corp., "Windows Debugging," <http://tinyurl.com/l6zsqzv>.
- [120] O. Mutlu, "The RowHammer Problem and Other Issues We May Face as Memory Becomes Denser," in *DATE*, 2017.
- [121] O. Mutlu, "Memory Scaling: A Systems Architecture Perspective," in *IMW*, 2013.
- [122] O. Mutlu, "Memory Scaling: A Systems Architecture Perspective," in *MEMCON*, 2013.
- [123] O. Mutlu and L. Subramanian, "Research Problems and Opportunities in Memory Systems," *SUPERFRI*, 2014.
- [124] P. J. Nair, D.-H. Kim, and M. K. Qureshi, "ArchShield: Architectural Framework for Assisting DRAM Scaling by Tolerating High Error Rates," in *MICRO*, 2013.
- [125] P. J. Nair, D. A. Roberts, and M. K. Qureshi, "Citadel: Efficiently Protecting Stacked Memory from TSV and Large Granularity Failures," *TACO*, vol. 12, no. 4, p. 49, 2016.
- [126] P. J. Nair, V. Sridharan, and M. K. Qureshi, "XED: Exposing On-Die Error Detection Information for Strong Memory Reliability," in *ISCA*, 2016.
- [127] R. Nishtala *et al.*, "Scaling Memcache at Facebook," in *NSDI*, 2013.
- [128] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, "Fast Crash Recovery in RAMCloud," in *SOSP*, 2011.
- [129] J. Ousterhout *et al.*, "The Case for RAMCloud," *Communications of the ACM*, vol. 54, no. 7, pp. 121–130, 2011.
- [130] T. Parnell, N. Papandreou, T. Mittelholzer, and H. Pozidis, "Modelling of the Threshold Voltage Distributions of Sub-20nm NAND Flash Memory," in *GLOBECON*, 2014.
- [131] M. Patel, J. Kim, and O. Mutlu, "The Reach Profiler (REAPER): Enabling the Mitigation of DRAM Retention Failures via Profiling at Aggressive Conditions," in *ISCA*, 2017.
- [132] K. Pattabiraman *et al.*, "Samurai: Protecting Critical Data in Unsafe Languages," in *EuroSys*, 2008.
- [133] A. J. Peña and P. Balaji, "Toward the Efficient Use of Multiple Explicitly Managed Memory Subsystems," in *CLUSTER*, 2014.
- [134] S. Phadke *et al.*, "MLP Aware Heterogeneous Memory System," in *DATE*, 2011.
- [135] M. Qureshi *et al.*, "AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems," in *DSN*, 2015.
- [136] M. K. Qureshi *et al.*, "Scalable High Performance Main Memory System Using Phase-Change Memory Technology," in *ISCA*, 2009.
- [137] L. E. Ramos, E. Gorbato, and R. Bianchini, "Page Placement in Hybrid Memory Systems," in *ICS*, 2011.
- [138] V. J. Reddi *et al.*, "Web Search Using Mobile Cores: Quantifying and Mitigating the Price of Efficiency," in *ISCA*, 2010.
- [139] A. C. Rieckstin *et al.*, "No More Electrical Infrastructure: Towards Fuel Cell Powered Data Centers," in *HotPower*, 2013.
- [140] H. Schirmeier *et al.*, "RAMpage: Graceful Degradation Management for Memory Errors in Commodity Linux Servers," in *PRDC*, 2011.
- [141] B. Schroeder *et al.*, "DRAM Errors in the Wild: A Large-Scale Field Study," in *SIGMETRICS Performance*, 2009.
- [142] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology," in *MICRO*, 2017.
- [143] V. Seshadri *et al.*, "RowClone: Fast and Efficient In-DRAM Copy and Initialization of Bulk Data," in *MICRO*, 2013.
- [144] T. Siddiqua *et al.*, "Analysis and Modeling of Memory Errors from Large-scale Field Data Collection," in *SELSE*, 2013.
- [145] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi, "Memory Errors in Modern Systems: The Good, The Bad, and the Ugly," in *ASPLOS*, 2015.
- [146] V. Sridharan *et al.*, "A Study of DRAM Failures in the Field," in *SC*, 2012.
- [147] V. Sridharan *et al.*, "Feng Shui of Supercomputer Memory: Positional Effects in DRAM and SRAM Faults," in *SC*, 2013.
- [148] J. Stuecheli *et al.*, "Elastic refresh: Techniques to mitigate refresh penalties in high density memory," in *MICRO*, 2010.
- [149] O. Subasi, G. Yalcin, F. Zylkyarov, O. Unsal, and J. Labarta, "Designing and Modelling Selective Replication for Fault-tolerant HPC Applications," in *CCGrid*, 2017.
- [150] D. Tang *et al.*, "Assessment of the Effect of Memory Page Retirement on System RAS Against Hardware Faults," in *DSN*, 2006.
- [151] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-Scale Cluster Management at Google with Borg," in *EuroSys*, 2015.
- [152] J. Wang, X. Zhu, Y. Liu, J. Zhang, M. Wu, W. Zhang, and K. Qiu, "Heterogeneous Energy-Efficient Cache Design in Warehouse Scale Computers," in *CF*, 2015.
- [153] Y.-M. Wang, Y. Huang, K.-P. Vo, P.-Y. Chung, and C. Kintala, "Checkpointing and Its Applications," in *FTCS*, 1995.

- [154] X. Xu *et al.*, "Understanding Soft Error Propagation Using Efficient Vulnerability-Driven Fault Injection," in *DSN*, 2012.
- [155] D. H. Yoon *et al.*, "Virtualized and Flexible ECC for Main Memory," in *ASPLOS*, 2010.
- [156] D. H. Yoon *et al.*, "BOOM: Enabling Mobile Memory Based Low-power Server DIMMs," in *ISCA*, 2012.
- [157] H. Yoon *et al.*, "Row Buffer Locality Aware Caching Policies for Hybrid Memories," in *ICCD*, 2012.
- [158] X. Yu, C. J. Hughes, N. Satish, O. Mutlu, and S. Devadas, "Banshee: Bandwidth-Efficient DRAM Caching via Software/Hardware Cooperation," in *MICRO*, 2017.
- [159] Y. Yue, "Caching in Datacenters," <https://twitter.github.io/pelikan/2016/04/03/caching-in-datacenters.html>, 2012.
- [160] W. Zhang and T. Li, "Exploring Phase Change Memory and 3D Die-Stacking for Power/Thermal Friendly, Fast and Durable Memory Architectures," in *PACT*, 2009.
- [161] R. Zheng and M. C. Huang, "Redundant Memory Array Architecture for Efficient Selective Protection," in *ISCA*, 2017.

A Memory Controller with Row Buffer Locality Awareness for Hybrid Memory Systems

HanBin Yoon^{1,2} Justin Meza^{3,2} Rachata Ausavarungnirun²
Rachael A. Harding^{4,2} Onur Mutlu^{5,2}

¹Google ²Carnegie Mellon University ³Facebook
⁴Massachusetts Institute of Technology ⁵ETH Zürich

This paper summarizes the idea and key contributions of the Dynamic Row Buffer Locality Aware Memory Controller (RBLA), which was published in ICCD 2012 [125], and examines the work’s significance and future potential. Non-volatile memory (NVM) is a class of promising scalable memory technologies that can potentially offer higher capacity than DRAM at the same cost point. Unfortunately, the access latency and energy of NVM is often higher than those of DRAM, while the endurance of NVM is lower. Many DRAM–NVM hybrid memory systems, also known as heterogeneous memory systems, use DRAM as a cache to NVM, to achieve the low access latency, low energy, and high endurance of DRAM, while taking advantage of the large capacity of NVM. A key question for a hybrid memory system is what data to cache in DRAM to best exploit the advantages of each technology while avoiding the disadvantages of each technology as much as possible.

We propose a new memory controller design that improves hybrid memory performance and energy efficiency. We observe that both DRAM and NVM banks employ row buffers that act as a cache for the most recently accessed memory row. Accesses that are row buffer hits incur similar latencies (and energy consumption) in both DRAM and NVM, whereas accesses that are row buffer misses incur longer latencies (and higher energy consumption) in NVM than in DRAM. To exploit this, we devise a policy that caches heavily-reused data that frequently misses in the NVM row buffers into DRAM. Our policy tracks the row buffer miss counts of recently-used rows in NVM, and caches in DRAM the rows that are predicted to incur frequent row buffer misses. Our proposed policy also takes into account the high write latencies of NVM, in addition to row buffer locality and more likely places the write-intensive pages in DRAM instead of NVM.

We evaluate our proposal using a hybrid memory consisting of DRAM and phase-change memory (PCM), a representative type of non-volatile memory. Compared to a conventional DRAM–PCM hybrid memory system that caches frequently-accessed data in DRAM, our row buffer locality-aware hybrid memory system improves average system performance by 14%, and average energy efficiency by 10%, on data-intensive server and cloud workloads. Our proposed hybrid memory system achieves a 31% performance gain over an all-PCM memory system, and comes within 29% of the performance of an all-

DRAM memory system (not taking PCM’s capacity benefit into account) on our evaluated workloads.

1. Introduction

Multiprogrammed and multithreaded workloads on chip multiprocessors require large amounts of main memory to support the working sets of many concurrently-executing threads. The demand for memory is increasing rapidly, as the number of cores or accelerators (collectively called *agents*) on a chip continues to increase and data-intensive applications become more widespread [35, 87, 91, 109]. *Dynamic Random Access Memory* (DRAM) is used to compose main memory in modern computers. Though strides in DRAM manufacturing process technology have enabled DRAM to scale to smaller feature sizes, and, thus, higher densities (capacity per unit area), it is predicted that DRAM density scaling will result in higher costs and lower reliability as the process technology feature size continues to decrease [19, 45, 50, 64, 75, 87, 88, 91, 99, 117]. Satisfying increasingly higher memory demands with exclusively DRAM will soon become too expensive in terms of both cost and energy.¹

1.1. Non-Volatile Memory

Emerging *non-volatile memory* (NVM) technologies such as *phase-change memory* (PCM) [55, 56, 57, 79, 97, 123, 126], *spin-transfer torque magnetic RAM* (STT-MRAM) [21, 38, 54, 92], *resistive RAM* (ReRAM) [24, 70, 110], and 3D XPoint [81], have shown promise for future main memory system designs to meet the increasing memory capacity demands of *data-intensive* workloads. With projected scaling trends, NVM cells can be manufactured more easily at smaller feature sizes than DRAM cells, achieving high density and capacity [21, 25, 26, 38, 54, 55, 56, 57, 70, 79, 92, 97, 99, 119, 123, 126, 131]. This is due to two reasons: (1) while a DRAM cell stores data in the form of charge, an NVM cell uses resistive values to represent the data, which is expected to scale to smaller feature sizes; and (2) unlike DRAM, several NVM devices use *multi-level cell* technology, which stores more than one bit of data per memory cell.

For example, PCM is a non-volatile memory technology that stores data by varying the electrical resistance of a material known as chalcogenide [55, 99, 123]. A PCM memory

¹We refer the reader to our prior works [17, 18, 19, 20, 39, 40, 48, 49, 50, 51, 52, 53, 61, 62, 63, 64, 65, 67, 68, 93, 105, 107] for a detailed background on DRAM.

cell is programmed by applying heat (via electrical current) to the chalcogenide and then cooling it at different rates, depending on the data to be stored. Rapid quenching places the chalcogenide into an amorphous state which has high resistance, representing the bit value of ‘0’ in single-level cell PCM, and slow cooling places the chalcogenide into a crystalline state which has low resistance, representing the bit value of ‘1’ in single-level cell PCM. Multi-level cell PCM can store multiple bits of data by providing more than two distinguishable resistance levels for each cell, very similar to the MLC NAND flash technology that is prevalent in modern storage systems [6, 7, 8, 9, 10, 11, 12, 12, 13, 14, 15, 16, 73, 96, 126].

However, NVM has a number of disadvantages. Compared to DRAM, NVM typically has a longer access latency, higher write energy, and lower endurance [55, 97]. For example, PCM’s long cooling duration required to crystallize chalcogenide leads to high PCM write latency, high read (sensing) latency, high read energy, and high write energy compared to those of DRAM [77]. Furthermore, the repeated thermal expansions and contractions of a PCM cell during programming lead to *finite write endurance*, which is estimated at 10^8 writes, an issue not present in DRAM [55].

1.2. Hybrid Memory Systems

Hybrid memory systems [1, 4, 5, 22, 29, 30, 34, 66, 76, 94, 95, 97, 100, 129] aim to combine the strengths of DRAM and emerging memory technologies (e.g., NVM, reduced-latency DRAM [64, 80, 104], reduced reliability DRAM [74, 97]). Many previous DRAM-NVM hybrid memory system designs employ DRAM as a small cache [97] or write buffer [29, 129] to NVM of large capacity. In this work, we utilize PCM to provide increased overall memory capacity (which leads to reduced page faults in the system), while the DRAM cache serves a large portion of the memory requests at low latency and low energy with high endurance. The combined effect increases overall system performance and energy efficiency [97]. A key question in the design of a DRAM-PCM hybrid memory system is how to place data between DRAM and PCM to best exploit the strengths of each technology while avoiding their weaknesses as much as possible.

1.3. Memory Device Architecture

In our ICCD 2012 paper [125], we develop new mechanisms for deciding how data should be placed in a DRAM-PCM hybrid memory system. Our main observation is that both DRAM and PCM devices consist of banks that employ row buffer circuitry. The organization of a memory bank is illustrated in Figure 1. Cells (memory elements) are typically laid out in arrays of rows (cells sharing a common *wordline*) and columns (cells sharing a common *bitline*). An access to the array occurs at the granularity of a row. To read from the array, a wordline is first asserted to select a row of cells. Then, through the bitlines, the contents of the selected cells are detected by sense amplifiers (labeled S/A in the figure) and latched by peripheral circuitry known as the *row buffer*.

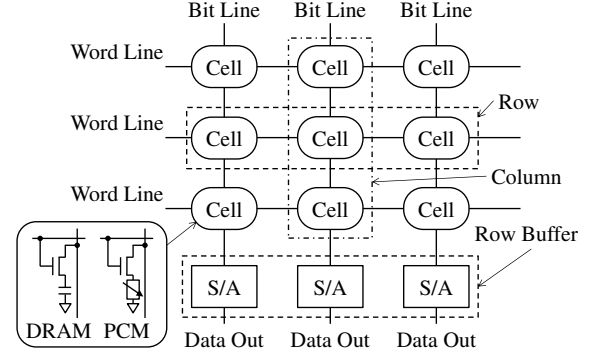


Figure 1: Memory cells organized in a 2D array of rows and columns. Reproduced from [125].

Once the contents of a row are latched in the row buffer, subsequent memory requests to that row are served promptly from the row buffer, without having to bear the delay of accessing the array. Such memory accesses are called *row buffer hits*. However, if a row different from the one latched in the row buffer is requested, then the newly requested row is read from the array into the row buffer (replacing the row buffer’s previous contents). Such a memory access incurs the high latency and energy of activating the array, and is called a *row buffer miss*. *Row buffer locality* (RBL) refers to the repeated reference to a row while its contents are in the row buffer. Memory requests to data with high row buffer locality are served efficiently (at low latency and energy) without having to frequently re-activate the memory cell array.

2. Row Buffer Locality-Aware Caching Policy

Our ICCD 2012 paper [125] proposes *Row Buffer Locality-Aware (RBLA)* caching policies, which a hybrid memory controller can use to guide data placement. RBLA can be used in any hybrid memory system where each underlying memory technology consists of banks with row buffers. We study an example hybrid memory system that consists of a large amount of PCM backed by a small DRAM cache [66, 72, 76, 97], whose organization is shown in Figure 2. Our main observation is that memory requests that *hit* in the row buffer incur *similar* latencies and energy consumption in both DRAM and PCM [55, 57], whereas requests that *miss* in the row buffer

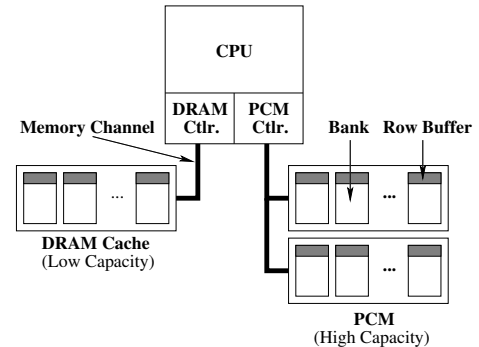


Figure 2: DRAM-PCM hybrid memory system organization. Reproduced from [125].

incur *higher* latency and energy in PCM than in DRAM. As a result, placing data that mostly leads to *row buffer hits* (i.e., data that has high row buffer locality) in DRAM provides *little benefit* over placing the same data in PCM. On the other hand, placing heavily reused data that leads to frequent row buffer misses (i.e., data that has low row buffer locality) in DRAM avoids the high latency and energy of PCM array accesses.

This observation is illustrated in the example shown in Figure 3. In the example, the service timelines for memory requests to rows A–D are shown. Prior hybrid memory and cache management proposals seek to improve the reuse of data placed in the cache and reduce the access bandwidth of the next level of memory (e.g., [43, 98]). We call this approach to cache management *conventional mapping*. Conventional mapping (top half of Figure 3) can place rows A and B (which have low row buffer locality) *both* in PCM, causing the high PCM array latency to become a bottleneck. In contrast, *row buffer locality-aware mapping* (bottom half of Figure 3) places rows A and B in DRAM such that they can benefit from DRAM’s lower array latency, leading to faster overall memory service.² Placing rows C and D (high row locality) in DRAM provides little benefit over placing them in PCM.

Based on this observation, we devise a hybrid memory caching policy that caches in DRAM the rows that mostly miss in the row buffer and are frequently reused. To implement this policy, the memory controller maintains a count of the row buffer misses for recently-used rows in PCM, and places in DRAM the data of rows whose row buffer miss counts exceed a certain threshold (dynamically adjusted at runtime in the RBLA-Dyn mechanism, which we describe in Section 2.3).

2.1. Measuring Row Buffer Locality

The RBLA mechanism tracks the row buffer locality statistics for a small number of recently-accessed rows, in a hardware structure called the *stats store*. The stats store resides in the memory controller, and is organized similarly to

²Even though the figure shows some requests being served in parallel, if the individual requests arrived in the same order at different times, the average request latency would still be improved significantly.

a cache, however its data payload per entry is a single row buffer miss counter.

On each PCM access, the memory controller looks for an entry in the stats store using the address of the accessed row. If there is no corresponding entry, a new entry is allocated for the accessed row, possibly evicting an older entry. If the access results in a row buffer miss, the row’s row buffer miss counter is incremented. If the access results in a row buffer hit, no additional action is taken.

2.2. Triggering Row Caching

Rows that exhibit low row buffer locality and high reuse will have high row buffer miss counter values. The RBLA mechanism selectively caches these rows by triggering the caching of a row in DRAM when the row’s row buffer miss counter exceeds a threshold value, *MissThresh*. Setting this *MissThresh* to a low value causes more rows with a higher row buffer locality to be cached.

Caching rows based on their row buffer locality attempts to migrate data between PCM and DRAM only when such data movement is beneficial. This affects system performance in three ways. First, placing in DRAM rows that have low row buffer locality improves average memory access latency, due to the lower row buffer miss latency of DRAM compared to PCM. Second, by selectively caching data that benefits from being migrated to DRAM, RBLA reduces unnecessary data movement between DRAM and PCM (i.e., data that frequently hits in the row buffer incurs the same access latency in PCM as in DRAM, and is thus left in PCM). This reduces memory bandwidth consumption, allowing more bandwidth to be used to serve demand requests, and enables better utilization of the DRAM cache space. Third, allowing data that frequently hits in the row buffer to remain in PCM contributes to balancing the memory request load between DRAM and PCM.

To prevent rows with low reuse from gradually building up large enough row buffer miss counts over an extended period of time to exceed *MissThresh* and trigger row caching, we

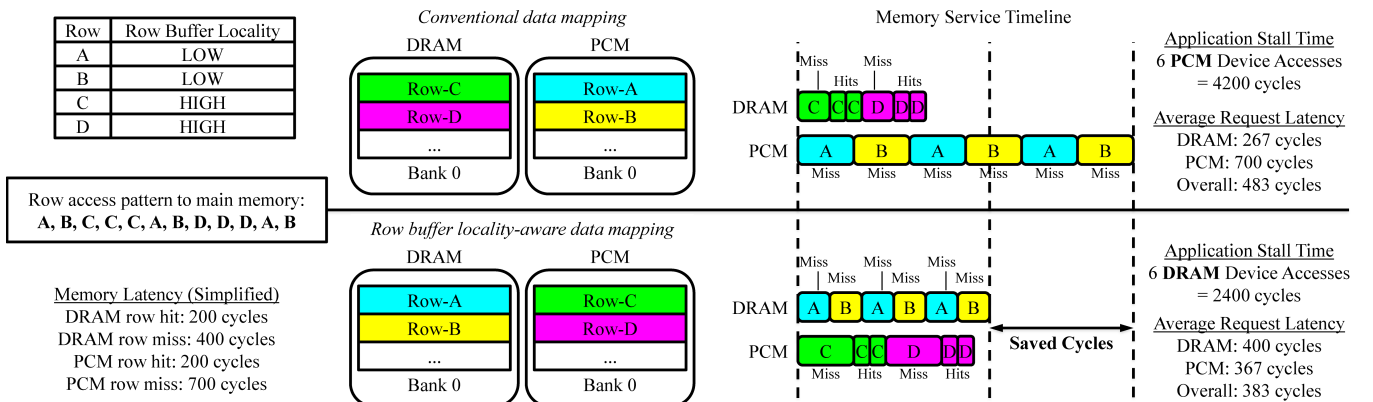


Figure 3: Conceptual example showing the importance of row buffer locality-awareness in hybrid memory data placement decisions. Reproduced from [125].

apply a periodic reset to all of the row buffer miss count values. We set this reset interval to 10 million cycles empirically.

2.3. Dynamic Threshold Adaptation: RBLA-Dyn

We improve the adaptivity of RBLA to workload and system variations by *dynamically* determining the value of MissThresh. The key idea behind this scheme, which we call RBLA-Dyn, is that *the number of cycles saved by caching rows in DRAM should outweigh the cost of migrating that data to DRAM*. RBLA-Dyn estimates, on an interval basis, the first order cost and benefit of employing a certain MissThresh value, and increases or decreases the MissThresh value to maximize the *net benefit* (i.e., benefit minus cost).

Since data migration operations can delay demand requests, we approximate cost as the number of cycles spent migrating each row across the memory channels ($t_{migration}$) times the number of rows migrated ($NumMigrations$):

$$Cost = NumMigrations \times t_{migration} \quad (1)$$

If these data migrations are eventually beneficial, the access latency to main memory will decrease. Hence, we can compute the benefit of migration as the number of cycles saved by accessing the data from the DRAM cache as opposed to PCM:

$$Benefit = NumReads_{drum} \times (t_{read,pcm} - t_{read,dram}) + \quad (2) \\ NumWrites_{drum} \times (t_{write,pcm} - t_{write,dram})$$

In this equation, $NumReads_{drum}$ and $NumWrites_{drum}$ are the number of reads and writes performed in DRAM after migration, $t_{read,dram}$ and $t_{write,dram}$ are the read and write latency of a DRAM row buffer miss, and $t_{read,pcm}$ and $t_{write,pcm}$ are the read and write latency of a PCM row buffer miss. RBLA-Dyn accounts for reads and writes separately, as they incur different latencies in many NVM technologies, such as PCM.

RBLA-Dyn uses a simple hill-climbing algorithm (see Algorithm 1 in our ICCD 2012 paper [125]) to find the value of MissThresh that maximizes the net benefit. The algorithm is executed at the end of each interval (10 million cycles in our setup). We refer the reader to Section IV-C of our ICCD 2012 paper [125] for more details on the RBLA-Dyn mechanism.

2.4. Implementation and Hardware Cost

The primary hardware cost incurred in implementing a row buffer locality-aware caching mechanism on top of an existing hybrid memory system is the stats store. We model a 16-way, 128-set, LRU-replacement stats store using 5-bit row buffer miss counters, which occupies a total of 9.25 KB. This stats store achieves within 0.3% of the system performance (and within 2.5% of the memory lifetime) of an unlimited-sized stats store for RBLA-Dyn.

3. Evaluation Methodology

We use a cycle-level in-house x86 multi-core simulator, whose front-end is based on Pin. The simulator is an early predecessor of Ramulator [53, 103] and the ThyNVM simulator [100]. We collect results using multiprogrammed workloads consisting of server- and cloud-type applications (including TPC-C/H [118], Apache Web Server, and video processing benchmarks) for a 16-core system. We compare our row buffer locality-aware caching policy (RBLA) against a policy that caches data that is frequently accessed (FREQ, similar in approach to [43]). We use this competitive baseline because we find that conventional LRU caching performs worse due to its high memory bandwidth consumption. FREQ caches a row when the number of accesses to the row exceeds a threshold value. FREQ-Dyn adopts the same dynamic threshold adjustment algorithm as RBLA-Dyn (Section 2.3). Our methodology and workloads are described in detail in Section VI of our ICCD 2012 paper [125].

4. Evaluation

Performance. Figure 4 shows the weighted speedup of the four caching techniques that we evaluate. As we observe from the figure, RBLA-Dyn provides the highest performance (14% improvement in weighted speedup over FREQ) among the four techniques. RBLA and RBLA-Dyn outperform FREQ and FREQ-Dyn, respectively, because the RBLA techniques place data with low row buffer locality in DRAM where it can be accessed at the lower DRAM array access latency, while keeping data with high row buffer locality in PCM where it can be accessed at the already-low row buffer hit latency.

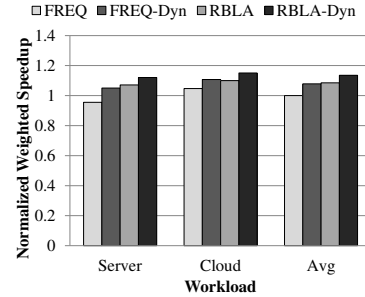


Figure 4: Weighted speedup of the four caching techniques: FREQ, FREQ-Dyn, RBLA, and RBLA-Dyn. Reproduced from [125].

Thread Fairness. Figure 5 shows the fairness of each caching technique. We measure fairness using *maximum slowdown* [3, 27, 28, 51, 52, 86, 112, 113, 115, 121, 122], which is the highest slowdown (reciprocal of speedup) experienced by any benchmark within the multiprogrammed workload. A *lower* maximum slowdown indicates *higher* fairness. We observe from the figure that RBLA-Dyn provides the highest thread fairness (6% improvement in maximum slowdown over FREQ) out of all evaluated policies. RBLA-Dyn throttles back on non-beneficial data migrations, reducing the amount of memory bandwidth and DRAM space consumed due to such

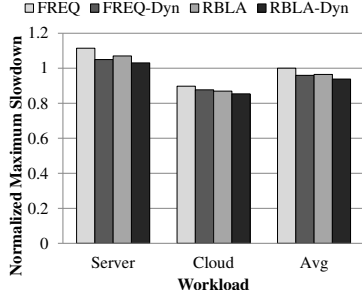


Figure 5: Fairness of the four caching techniques: FREQ, FREQ-Dyn, RBLA, and RBLA-Dyn (lower is better). Reproduced from [125].

migrations. Combined with the reduced average memory access latency, RBLA-Dyn reduces contention for memory bandwidth among co-running applications, providing higher fairness.

Memory Energy Efficiency. Figure 6 shows that RBLA-Dyn achieves the highest memory energy efficiency (10% improvement over FREQ) compared to other policies, in terms of performance per Watt. This is because RBLA-Dyn places data with low row buffer locality in DRAM, making the energy cost of row buffer miss accesses lower than it would be if such data were placed in PCM. RBLA-Dyn also reduces energy consumption by reducing the amount of non-beneficial or useless data migrations.

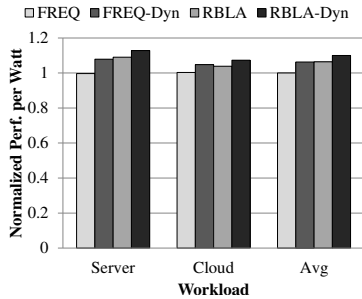


Figure 6: Energy efficiency of the four caching techniques: FREQ, FREQ-Dyn, RBLA, and RBLA-Dyn. Reproduced from [125].

We provide the following other evaluation results in Section VII of our ICCD 2012 paper [125]:

- Impact of RBLA-Dyn on average memory latency (Section VII-A of [125]).
- Impact of RBLA-Dyn on DRAM and PCM channel utilization (Section VII-A of [125]).
- Memory access breakdown of each workload to DRAM and PCM (Section VII-A of [125]).
- Impact of RBLA-Dyn on PCM lifetime (Section VII-D of [125]).
- Comparison with all-PCM and all-DRAM systems (Section VII-E of [125]).

As we discuss in detail in our ICCD 2012 paper [125], RBLA-Dyn bridges the gap in performance between homogeneous all-DRAM and all-PCM memory systems of equal addressable capacity (achieving within 29% of the performance of an all-

DRAM system, and improving performance by 31% over an all-PCM system), while providing close to seven years of memory lifetime.³

We conclude that taking row buffer locality into account enables new hybrid memory caching policies that achieve high performance and energy efficiency.

5. Related Work

To our knowledge, our ICCD 2012 paper [125] is the first work to observe that row buffer hit latencies are similar in different memory technologies, and uses this observation to devise a caching policy that improves the performance and energy efficiency of a hybrid memory system. No previous work, as far as we know, considered row buffer locality as a key metric for deciding what data to cache and what not to cache. We discuss related work on caching policies and hybrid memory systems.

Caching Based on Data Access Frequency. Jiang et al. [43] propose caching only the data that experiences a high number of accesses in an on-chip DRAM cache (in 4–8 KB block sizes), to reduce off-chip memory bandwidth consumption. Johnson and Hwu [44] use a counter-based mechanism to track data reuse at a granularity larger than a cache block. Cache blocks in a region with less reuse bypass a direct-mapped cache if that region conflicts with another that has more reuse. We propose to take advantage of row buffer locality in memory banks when employing off-chip DRAM and PCM. We exploit the fact that accesses to DRAM and PCM have similar average latencies for rows that have high row buffer locality.

Ramos et al. [98] adapt a buffer cache replacement algorithm to rank pages based on their frequency and recency of accesses, and place the highly-ranking pages in DRAM, in a DRAM-PCM hybrid memory system. Our work is orthogonal, because the page-ranking algorithm can be adapted to rank pages based on their frequency and recency of row buffer misses (not counting accesses that are row buffer hits), for which we expect improved performance.

Caching Based on Locality of Data Access. Gonzalez et al. [37] propose placing data in one of two last-level caches depending on whether it exhibits spatial or temporal locality. They also propose bypassing the cache when accessing large data structures with large strides (e.g., big matrices) to prevent cache thrashing. Rivers and Davidson [101] propose separating out data without temporal locality from data with, and placing it in a special buffer to prevent the pollution of the L1 cache. These works are primarily concerned with on-chip L1/L2 caches that have access latencies on the order of a few to tens of processor clock cycles, where off-chip memory bank row buffer locality is less applicable.

³Note that lifetime can be further improved by enabling more aggressive write optimization [106], and by taking advantage of application-level error tolerance [74].

There have been many works in on-chip caching to improve cache utilization (e.g., a recent one uses an evicted address filter to predict cache block reuse [108]), but none of these consider the row buffer locality of cache misses.

Caching Based on Other Criteria. Chatterjee et al. [22] observe that the first word of cache blocks is critical to performance, and propose to store only the first word of each block in fast DRAM. Phadke and Narayanasamy [95] propose to classify applications into three categories based on memory-level parallelism (MLP): latency-sensitive, bandwidth-sensitive, and insensitive-to-both. To estimate MLP, they profile the misses per kilo-instruction (MPKI) and stall time of each application offline during the compilation stage. Applications with high MPKI but low stall time are considered to have good MLP.

Hybrid Memory Systems. Qureshi et al. [97] propose increasing the size of main memory by adopting PCM, and using DRAM as a conventional cache to PCM. The reduction in page faults due to the increase in main memory size brings performance and energy improvements to the system. Our ICCD 2012 paper [125] proposes a new, effective DRAM caching policy to PCM, and studies performance effects without page faults present.

Li et al. [66] propose UHM, a utility-based hybrid memory management mechanism that expands upon our RBLA policy. UHM estimates the *utility* of each page, which is the benefit to system performance of placing each page in different types of memory (e.g., DRAM and NVM). UHM migrates to the fast memory of a hybrid memory system only those pages whose utility would improve the most after migration.

Ren et al. [100] propose ThyNVM, which manages the DRAM and PCM spaces carefully and adapts the granularity of management to the access patterns in a manner that provides crash consistency in a persistent memory system.

Dhiman et al. [29] propose a hybrid main memory system that exposes DRAM and PCM addressability to the software (OS). If the number of writes to a particular PCM page exceeds a certain threshold, the contents of the page are copied to another page (either in DRAM or PCM), thus facilitating PCM wear-leveling. Mogul et al. [82] suggest that the OS exploit metadata information available to it to make data placement decisions between DRAM and non-volatile memory. Similar to [29], their data placement criteria are centered around the write frequency to data. Our proposal is complementary to this work, and row buffer locality information, if exposed, can be used by the OS to place pages in DRAM or PCM.

Bivens et al. [4] examine the various design concerns of a heterogeneous memory system such as memory latency, bandwidth, and endurance requirements of employing storage class memory (e.g., PCM, STT-MRAM, NAND flash memory). Their hybrid memory organization is similar to ours and that in [97], in that DRAM is used as a cache to a slower memory medium, transparently to software. Phadke et al. [95] propose to profile the memory access patterns

of individual applications in a multi-core system, and place their working sets in the particular type of DRAM that best suits the application’s memory demands. In contrast, RBLA dynamically makes fine-grained data placement decisions at a row granularity, depending on the row buffer locality characteristics of each page.

Agarwal et al. [1] propose a software-based approach to manage huge pages (e.g., 2MB pages) in hybrid memory systems. The mechanism profiles the memory access patterns of huge pages, and uses the profiling information to guide page migration between DRAM and NVM. Peña and Balaji [94] propose a profiling tool to assess the impact of distributing memory objects across memory devices in hybrid memory systems. Bock et al. [5] propose a scheme that allows concurrent migration of multiple pages between different types of memory devices without significantly affecting the memory bandwidth. Gai et al. [34] propose a data placement scheme that aims to minimize the energy consumption of hybrid memory systems. Liu et al. [69] propose a scheme that manages the entire memory hierarchy, which includes caches, memory channels, and DRAM/NVM banks. Dulloor et al. [30] propose a programmer-guided data placement tool, which requires programmers to modify the source code, and needs data from a representative profiling run of the application, prior to making placement decisions. Ideas from all of these works can be combined with RBLA for better performance and efficiency.

Exploiting Row Buffer Locality. Many previous works exploit row buffer locality to improve memory system performance, but none (to our knowledge) develop a cache data placement policy that considers the row buffer locality of the block to be cached. Lee et al. [55, 56, 57] propose to use multiple short row buffers in PCM devices, much like an internal device cache, to increase the row buffer hit rate. Meza et al. [77] examine the case for small row buffers for NVM devices. Sudan et al. [116] propose a mechanism that identifies frequently referenced sub-rows of data, and migrates them to reserved rows. By co-locating these frequently accessed sub-rows, this scheme aims to increase the row buffer hit rate of memory accesses, and improve performance and energy consumption. DRAM-aware last-level cache writeback schemes [60, 111] speculatively issue writeback requests that are predicted to hit in the row buffer. RBLA is complementary to these works, and can be applied together with them because RBLA targets a different problem.

Row buffer locality is also commonly exploited in memory scheduling algorithms. The First-Ready First-Come-First-Serve algorithm (FR-FCFS) [102, 132] prioritizes memory requests that hit in the row buffer, improving the latency, throughput, and energy cost of serving memory requests. Many other memory scheduling algorithms [3, 31, 32, 33, 36, 41, 42, 46, 47, 51, 52, 58, 59, 60, 71, 83, 84, 85, 86, 89, 90, 111, 112, 113, 114, 115, 120, 121, 124, 130] build upon this “row-hit first” principle.

Muralidhara et al. [86] use a thread’s row buffer locality as a metric to decide which channel the thread’s pages should be allocated to in a multi-channel memory system. Their goal is to reduce memory interference between threads, and as such their technique is complementary to ours.

6. Significance

Our ICCD 2012 paper [125] makes several novel contributions that we expect will have a long-term impact on the design of memory systems, and we believe that our work inspires several new research questions.

6.1. Long-Term Impact

The memory scaling bottleneck continues to be a significant hurdle to system performance and energy efficiency [87, 88, 91]. Emerging applications operate on increasingly-larger sets of data, and require high-capacity, high-performance main memories, but the poor scaling of DRAM limits the ability of these applications to fit their entire working sets within a DRAM-based main memory. Because DRAM cannot keep pace with application needs, we expect that the demand for alternative memory technologies will continue to grow in the coming years.

Hybrid memory systems can allow systems to harness these alternative memory technologies without fully sacrificing the benefits of DRAM. By combining slower but larger memories (e.g., NVM) with faster but smaller memories (e.g., DRAM), a hybrid memory system has the potential to provide the illusion of a fast and large memory system at a reasonable cost. However, as we discuss, this potential can only be realized by carefully considering which pieces of data are placed in each of the constituent memories of a hybrid memory system. To our knowledge, our ICCD 2012 paper [125] is the first to show that the organization of the underlying memory technologies, such as the existence of row buffers, can be used to make more intelligent data placement decisions.

While our ICCD 2012 paper [125] shows the impact of our proposed data placement policy on a hybrid memory consisting of DRAM and PCM, it can be used to enable a wide range of hybrid memory systems. For example, STT-MRAM devices can make use of a row buffer [2, 54, 77, 78], and expensive reduced-latency DRAM devices [80, 104] also make use of a row buffer. RBLA can be used to improve the performance of hybrid memories that include any of these memory technologies, as our general observations on row buffer locality remain the same. We expect that this versatility will increase the potential impact of RBLA, as no single memory technology has yet to emerge as the dominant replacement for DRAM.

6.2. Research Questions

As we show in our ICCD 2012 paper [125], the efficient management of hybrid memory systems requires the identification and consideration of the key similarities and trade-offs

of each memory type. An open research question inspired by RBLA’s use of row buffer locality is *what other properties of memory systems should hybrid memory management mechanisms consider?* For example, one of our recent works [66] incorporates information on memory-level parallelism (MLP) into data placement decisions in hybrid memory management. As that work shows, we can use a combination of access frequency, row buffer locality, and MLP to predict the overall performance impact of migrating a page between each memory type. As future memory technologies are developed, we expect that other such properties will be important to consider, in order to maximize the benefits provided by the hybrid memory system.

Several works propose on-chip DRAM caches [23, 43, 127, 128], where a small amount of DRAM is used as a last-level cache to reduce the number of accesses to a larger off-chip DRAM. This is akin to the design of a hybrid memory system, but there are different trade-offs in each design. For example, while the row buffer hit latency is typically similar across memory technologies in hybrid memories, both a row buffer hit and a row buffer miss take longer when accessing the off-chip DRAM as opposed to accessing the on-chip DRAM cache. This inspires us to ask *how can principles of hybrid memory systems be applied to DRAM cache management, and vice versa?* Extending upon this, *can we design general mechanisms that can be applied to both hybrid memory systems and DRAM cache management?* As one example, our recent work [66] on predicting the utility of data placement decisions is highly parameterized, and these parameters can easily be tuned to represent the trade-offs in both hybrid memory systems and in systems with a DRAM cache. We believe and hope that future works should strive to develop other such general mechanisms.

7. Conclusion

Our ICCD 2012 paper [125] observes that row buffer access latency (and energy) in DRAM and PCM are similar, while PCM array access latency (and energy) is much higher than DRAM array access latency (and energy). Therefore, in a hybrid memory system where DRAM is used as a cache to PCM, it makes sense to place in DRAM data that would cause frequent row buffer misses as such data, if placed in PCM, would incur the high PCM array access latency. We develop a caching policy that achieves this effect by keeping track of rows that have high row buffer miss counts (i.e., low row buffer locality, but high reuse) and places only such rows in DRAM. Our final policy dynamically determines the threshold used to decide whether a row has low locality based on cost-benefit analysis. Evaluations show that the proposed row buffer locality aware caching policy provides better performance, fairness, and energy-efficiency compared to caching policies that only consider access frequency or recency. Our mechanisms are applicable to and can improve the performance of other hybrid memory systems consisting

of different technologies. We hope that our findings can help ease the adoption of emerging memory technologies in future systems, and inspire further research in data management policies.

Acknowledgments

We thank Saugata Ghose for his dedicated effort in the preparation of this article. We acknowledge the support of AMD, HP, Intel, Oracle, and Samsung. This research was partially supported by the NSF (grants 0953246 and 1212962), GSRC, Intel URO, and ISTC on Cloud Computing. HanBin Yoon was partially supported by the Samsung Scholarship.

References

- [1] N. Agarwal and T. F. Wenisch, "Thermostat: Application-transparent Page Management for Two-tiered Main Memory," in *ASPLOS*, 2017.
- [2] T. W. Andre, J. J. Nahas, C. K. Subramanian, B. J. Garni, H. S. Lin, A. Omair, and W. L. Martino, Jr., "A 4-Mb 0.18- μ m 1T1MTJ Toggle MRAM with Balanced Three Input Sensing Scheme and Locally Mirrored Unidirectional Write Drivers," *JSSC*, 2005.
- [3] R. Ausavarungnirun, K. Chang, L. Subramanian, G. Loh, and O. Mutlu, "Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems," in *ISCA*, 2012.
- [4] A. Bivens *et al.*, "Architectural design for next generation heterogeneous memory systems," in *IMW*, 2010.
- [5] S. Bock, B. R. Childers, R. Melhem, and D. Mossé, "Concurrent Migration of Multiple Pages in Software-Managed Hybrid Main Memory," in *ICCD*, 2016.
- [6] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, and O. Mutlu, "Error Characterization, Mitigation, and Recovery in Flash-Memory-Based Solid-State Drives," *Proc. IEEE*, 2017.
- [7] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, and O. Mutlu, "Error Characterization, Mitigation, and Recovery in Flash Memory Based Solid-State Drives," arXiv:1706.08642 [cs.AR], 2017.
- [8] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, and O. Mutlu, "Errors in Flash-Memory-Based Solid-State Drives: Analysis, Mitigation, and Recovery," arXiv:1711.11427 [cs.AR], 2017.
- [9] Y. Cai, S. Ghose, Y. Luo, K. Mai, O. Mutlu, and E. F. Haratsch, "Vulnerabilities in MLC NAND Flash Memory Programming: Experimental Analysis, Exploits, and Mitigation Techniques," in *HPCA*, 2017.
- [10] Y. Cai, E. F. Haratsch, O. Mutlu, and K. Mai, "Error Patterns in MLC NAND Flash Memory: Measurement, Characterization, and Analysis," in *DATE*, 2012.
- [11] Y. Cai, Y. Luo, E. F. Haratsch, K. Mai, and O. Mutlu, "Data Retention in MLC NAND Flash Memory: Characterization, Optimization, and Recovery," in *HPCA*, 2015.
- [12] Y. Cai, G. Yalcin, O. Mutlu, E. F. Haratsch, A. Cristal, O. Unsal, and K. Mai, "Flash Correct and Refresh: Retention Aware Management for Increased Lifetime," in *ICCD*, 2012.
- [13] Y. Cai, G. Yalcin, O. Mutlu, E. F. Haratsch, A. Cristal, O. Unsal, and K. Mai, "Error Analysis and Retention-Aware Error Management for NAND Flash Memory," *Intel Technology Journal*, 2013.
- [14] Y. Cai, G. Yalcin, O. Mutlu, E. F. Haratsch, O. Unsal, A. Cristal, and K. Mai, "Neighbor Cell Assisted Error Correction in MLC NAND Flash Memories," in *SIGMETRICS*, 2014.
- [15] Y. Cai, E. F. Haratsch, O. Mutlu, and K. Mai, "Threshold Voltage Distribution in MLC NAND Flash Memory: Characterization, Analysis, and Modeling," in *DATE*, 2013.
- [16] Y. Cai, Y. Luo, S. Ghose, E. F. Haratsch, K. Mai, and O. Mutlu, "Read Disturb Errors in MLC NAND Flash Memory: Characterization, Mitigation, and Recovery," in *DSN*, 2015.
- [17] K. Chang, A. Yaglikci, S. Ghose, A. Agrawal, N. Chatterjee, A. Kashyap, D. Lee, M. O'Connor, H. Hassan, and O. Mutlu, "Understanding Reduced-Voltage Operation in Modern DRAM Devices: Experimental Characterization, Analysis, and Mechanisms," in *SIGMETRICS*, 2017.
- [18] K. K. Chang, D. Lee, Z. Chishti, A. R. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu, "Improving DRAM Performance by Parallelizing Refreshes with Accesses," in *HPCA*, 2014.
- [19] K. Chang, A. Kashyap, H. Hassan, S. Ghose, K. Hsieh, D. Lee, T. Li, G. Pekhimenko, S. Khan, and O. Mutlu, "Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization," in *SIGMETRICS*, 2016.
- [20] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu, "Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM," in *HPCA*, 2016.
- [21] M. T. Chang, P. Rosenfeld, S. L. Lu, and B. Jacob, "Technology Comparison for Large Last-Level Caches (L3Cs): Low-Leakage SRAM, Low Write-Energy STT-RAM, and Refresh-Optimized eDRAM," in *HPCA*, 2013.
- [22] N. Chatterjee, M. Shevgoor, R. Balasubramanian, A. Davis, Z. Fang, R. Illikkal, and R. Iyer, "Leveraging Heterogeneity in DRAM Main Memories to Accelerate Critical Word Access," in *MICRO*, 2012.
- [23] C. 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 *MICRO*, 2014.
- [24] L. Chua, "Memristor—The Missing Circuit Element," *TCT*, Sep. 1971.
- [25] K. C. Chun, H. Zhao, J. Harms, T.-H. Kim, J.-P. Wang, and C. Kim, "A Scaling Roadmap and Performance Evaluation of In-Plane and Perpendicular MTJ Based STT-MRAMs for High-Density Cache Memory," *JSSC*, 2013.
- [26] S. Chung *et al.*, "Fully Integrated 54nm STT-RAM with the Smallest Bit Cell Dimension for High Density Memory Application," in *IEDM*, 2010.
- [27] R. Das, R. Ausavarungnirun, O. Mutlu, A. Kumar, and M. Azimi, "Application-to-core Mapping Policies to Reduce Memory System Interference in Multi-core Systems," in *HPCA*, 2013.
- [28] R. Das, O. Mutlu, T. Moscibroda, and C. R. Das, "Application-Aware Prioritization Mechanisms for On-Chip Networks," in *MICRO*, 2009.
- [29] G. Dhiman *et al.*, "PDRAM: a Hybrid PRAM and DRAM Main Memory System," in *DAC*, 2009.
- [30] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan, "Data Tiering in Heterogeneous Memory Systems," in *EuroSys*, 2016.
- [31] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Fairness via Source Throttling: A Configurable and High-performance Fairness Substrate for Multi-core Memory Systems," in *ASPLOS*, 2010.
- [32] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Prefetch-aware Shared Resource Management for Multi-core Systems," in *ISCA*, 2011.
- [33] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt, "Parallel Application Memory Scheduling," in *MICRO*, 2011.
- [34] K. Gai, M. Qiu, H. Zhao, and L. Qiu, "Smart Energy-Aware Data Allocation for Heterogeneous Memory," in *HPCC/SmartCity/DSS*, 2016.
- [35] W. Gao, L. Wang, J. Zhan, C. Luo, D. Zheng, Z. Jia, B. Xie, C. Zheng, Q. Yang, and H. Wang, "A Dwarf-based Scalable Big Data Benchmarking Methodology," *arXiv CoRR*, 2017.
- [36] S. Ghose, H. Lee, and J. F. Martinez, "Improving Memory Scheduling via Processor-side Load Criticality Information," in *ISCA*, 2013.
- [37] A. González *et al.*, "A data cache with multiple caching strategies tuned to different types of locality," in *ICS*, 1995.
- [38] X. Guo, E. İpek, and T. Soyata, "Resistive Computation: Avoiding the Power Wall with Low-Leakage, STT-MRAM Based Computing," in *ISCA*, 2009.
- [39] H. Hassan, N. Vijaykumar, S. Khan, S. Ghose, K. Chang, G. Pekhimenko, D. Lee, O. Ergin, and O. Mutlu, "SoftMC: A Flexible and Practical Open-Source Infrastructure for Enabling Experimental DRAM Studies," in *HPCA*, 2017.
- [40] H. Hassan, G. Pekhimenko, N. Vijaykumar, V. Seshadri, D. Lee, O. Ergin, and O. Mutlu, "ChargeCache: Reducing DRAM Latency by Exploiting Row Access Locality," in *HPCA*, 2016.
- [41] E. İpek, O. Mutlu, J. F. Martinez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," in *ISCA*, 2008.
- [42] M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver, "A QoS-Aware Memory Controller for Dynamically Balancing GPU and CPU Bandwidth Use in an MPSoC," in *DAC*, 2012.
- [43] X. Jiang *et al.*, "CHOP: Adaptive Filter-Based DRAM Caching for CMP Server Platforms," in *HPCA*, 2010.
- [44] T. L. Johnson and W.-m. Hwu, "Run-time adaptive cache hierarchy management via reference analysis," in *ISCA*, 1997.
- [45] U. Kang, H.-S. Yu, C. Park, H. Zheng, J. Halbert, K. Bains, S. Jang, and J. Choi, "Co-Architecting Controllers and DRAM to Enhance DRAM Process Scaling," in *The Memory Forum*, 2014.
- [46] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, "Bounding Memory Interference Delay in COTS-based Multi-core Systems," in *RTAS*, 2014.
- [47] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, "Bounding and Reducing Memory Interference in COTS-based Multi-core Systems," *Real-Time Systems*, vol. 52, no. 3, May 2016.
- [48] J. S. Kim, M. Patel, H. Hassan, and O. Mutlu, "The DRAM Latency PUF: Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency-Reliability Tradeoff in Modern DRAM Devices," in *HPCA*, 2018.
- [49] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM," in *ISCA*, 2012.
- [50] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," in *ISCA*, 2014.
- [51] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers," in *HPCA*, 2010.
- [52] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *MICRO*, 2010.

- [53] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," *CAL*, 2015.
- [54] E. Kultursay, M. Kandemir, A. Sivasubramanian, and O. Mutlu, "Evaluating STT-RAM as an Energy-Efficient Main Memory Alternative," in *ISPASS*, 2013.
- [55] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting Phase Change Memory As a Scalable DRAM Alternative," in *ISCA*, 2009.
- [56] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Phase Change Memory Architecture and the Quest for Scalability," *Communications of the ACM*, 2010.
- [57] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-Change Technology and the Future of Main Memory," *IEEE Micro*, vol. 30, January 2010.
- [58] C. J. Lee, V. Narasiman, O. Mutlu, and Y. N. Patt, "Improving Memory Bank-Level Parallelism in the Presence of Prefetching," in *MICRO*, 2009.
- [59] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt, "Prefetch-aware DRAM Controllers," in *MICRO*, 2008.
- [60] 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," Univ. of Texas at Austin, High Performance Systems Group, Tech. Rep. TR-HPS-2010-002, 2010.
- [61] D. Lee, S. Khan, L. Subramanian, S. Ghose, R. Ausavarungnirun, G. Pekhimenko, V. Seshadri, and O. Mutlu, "Design-Induced Latency Variation in Modern DRAM Chips: Characterization, Analysis, and Latency Reduction Mechanisms," in *SIGMETRICS*, 2017.
- [62] D. Lee, S. Ghose, G. Pekhimenko, S. Khan, and O. Mutlu, "Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost," *TACO*, 2016.
- [63] D. Lee, Y. Kim, G. Pekhimenko, S. Khan, V. Seshadri, K. Chang, and O. Mutlu, "Adaptive-Latency DRAM: Optimizing DRAM Timing for the Common-Case," in *HPCA*, 2015.
- [64] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu, "Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture," in *HPCA*, 2013.
- [65] D. Lee, L. Subramanian, R. Ausavarungnirun, J. Choi, and O. Mutlu, "Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM," in *PACT*, 2015.
- [66] Y. Li, S. Ghose, J. Choi, J. Sun, H. Wang, and O. Mutlu, "Utility-Based Hybrid Memory Management," in *CLUSTER*, 2017.
- [67] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, "RAIDR: Retention-Aware Intelligent DRAM Refresh," in *ISCA*, 2012.
- [68] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu, "An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms," in *ISCA*, 2013.
- [69] L. Liu, H. Yang, Y. Li, M. Xie, L. Li, and C. Wu, "Memos: A Full Hierarchy Hybrid Memory Management Framework," in *ICCD*, 2016.
- [70] T. Liu *et al.*, "A 130.7mm² 2-Layer 32Gb ReRAM Memory Device in 24nm Technology," *JSSC*, 2014.
- [71] W. Liu, P. Huang, T. Kun, T. Lu, K. Zhou, C. Li, and X. He, "LAMS: A Latency-aware Memory Scheduling Policy for Modern DRAM Systems," in *IPCCC*, 2016.
- [72] G. H. Loh and M. D. Hill, "Efficiently enabling conventional block sizes for very large die-stacked DRAM caches," in *MICRO*, 2011.
- [73] Y. Luo, S. Ghose, Y. Cai, E. F. Haratsch, and O. Mutlu, "Enabling Accurate and Practical Online Flash Channel Modeling for Modern MLC NAND Flash Memory," *JSAC*, 2016.
- [74] Y. Luo, S. Govindan, B. Sharma, M. Santaniello, J. Meza, A. Kansal, J. Liu, B. Khes-sib, K. Vaid, and O. Mutlu, "Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-reliability Memory," in *DSN*, 2014.
- [75] J. A. Mandelman *et al.*, "Challenges and Future Directions for the Scaling of Dynamic Random-Access Memory (DRAM)," *IBM JRD*, vol. 46, 2002.
- [76] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan, "Enabling Efficient and Scalable Hybrid Memories Using Fine-Granularity DRAM Cache Management," *CAL*, 2012.
- [77] J. Meza, J. Li, and O. Mutlu, "A Case for Small Row Buffers in Non-Volatile Main Memories," in *ICCD Poster Session*, 2012.
- [78] J. Meza, J. Li, and O. Mutlu, "Evaluating Row Buffer Locality in Future Non-Volatile Main Memories," Carnegie Mellon Univ., SAFARI Research Group, Tech. Rep. TR-SAFARI-2012-002, 2012.
- [79] J. Meza, Y. Luo, S. Khan, J. Zhao, Y. Xie, and O. Mutlu, "A Case for Efficient Hardware/software Cooperative Management of Storage and Memory," in *WEED*, 2013.
- [80] Micron Technology, Inc., "576Mb: x18, x36 RLD RAM3," 2011.
- [81] Micron Technology, Inc., "Breakthrough Nonvolatile Memory Technology," <https://www.micron.com/about/our-innovation/3d-xpoint-technology>, 2016.
- [82] J. C. Mogul *et al.*, "Operating system support for NVM+DRAM hybrid main memory," in *HotOS*, 2009.
- [83] T. Moscibroda and O. Mutlu, "Memory Performance Attacks: Denial of Memory Service in Multi-core Systems," in *USENIX Security*, 2007.
- [84] T. Moscibroda and O. Mutlu, "Distributed Order Scheduling and Its Application to Multi-core DRAM Controllers," in *PODC*, 2008.
- [85] J. Mukundan and J. F. Martinez, "MORSE: Multi-objective Reconfigurable Self-optimizing Memory Scheduler," in *HPCA*, 2012.
- [86] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, "Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning," in *MICRO*, 2011.
- [87] O. Mutlu, "Memory scaling: A systems architecture perspective," in *IMW*, 2013.
- [88] O. Mutlu, "The RowHammer Problem and Other Issues We May Face as Memory Becomes Denser," in *DATE*, 2017.
- [89] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *MICRO*, 2007.
- [90] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems," in *ISCA*, 2008.
- [91] O. Mutlu and L. Subramanian, "Research Problems and Opportunities in Memory Systems," *SUPERFRI*, 2015.
- [92] H. Naeimi, C. Augustine, A. Raychowdhury, S.-L. Lu, and J. Tschanz, "STT-RAM Scaling and Retention Failure," *Intel Technol. J.*, May 2013.
- [93] M. Patel, J. Kim, and O. Mutlu, "The Reach Profiler (REAPER): Enabling the Mitigation of DRAM Retention Failures via Profiling at Aggressive Conditions," in *ISCA*, 2017.
- [94] A. J. Peña and P. Balaji, "Toward the Efficient Use of Multiple Explicitly Managed Memory Subsystems," in *CLUSTER*, 2014.
- [95] S. Phadke and S. Narayanasamy, "MLP Aware Heterogeneous Memory System," in *DATE*, 2011.
- [96] M. K. Qureshi, M. M. Franceschini, L. A. Lastras-Montano, and J. P. Karidis, "Morphable Memory System: A Robust Architecture for Exploiting Multi-level Phase Change Memories," in *ISCA*, 2010.
- [97] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable High Performance Main Memory System using Phase-change Memory Technology," in *ISCA*, 2009.
- [98] L. E. Ramos *et al.*, "Page placement in hybrid memory systems," in *JCS*, 2011.
- [99] S. Raoux *et al.*, "Phase-Change Random Access Memory: A Scalable Technology," *IBM JRD*, 2008.
- [100] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu, "ThyNVM: Enabling software-transparent crash consistency in persistent memory systems," in *MICRO*, 2015.
- [101] J. Rivers and E. Davidson, "Reducing conflicts in direct-mapped caches with a temporality-based design," in *ICPP*, 1996.
- [102] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory Access Scheduling," in *ISCA*, 2000.
- [103] SAFARI Research Group, "Ramulator: A DRAM Simulator – GitHub Repository," <https://github.com/CMU-SAFARI/ramulator>.
- [104] Y. Sato *et al.*, "Fast cycle RAM (FCRAM): A 20-ns Random Row Access, Pipeline-Operating DRAM," in *VLSIC*, 1998.
- [105] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology," in *MICRO*, 2017.
- [106] V. Seshadri, A. Bhowmick, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "The Dirty-Block Index," in *ISCA*, 2014.
- [107] V. Seshadri *et al.*, "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization," in *MICRO*, 2013.
- [108] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry, "The Evicted-Address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing," in *PACT*, 2012.
- [109] M. Silva, M. R. Hines, D. Gallo, Q. Liu, K. D. Ryu, and D. da Silva, "CloudBench: Experiment Automation for Cloud Environments," in *IC2E*, 2013.
- [110] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The Missing Memristor Found," *Nature*, May 2008.
- [111] 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*, 2010.
- [112] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, "BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling," in *IEEE TPDS*, 2016.
- [113] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, "The Blacklisting Memory Scheduler: Achieving high performance and fairness at low cost," in *ICCD*, 2014.
- [114] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu, "The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-application Interference at Shared Caches and Main Memory," in *MICRO*, 2015.
- [115] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu, "MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems," in *HPCA*, 2013.
- [116] K. Sudan *et al.*, "Micro-pages: increasing DRAM efficiency with locality-aware data placement," in *ASPLOS*, 2010.
- [117] The International Technology Roadmap for Semiconductors, "Process integration, devices, and structures," 2010.
- [118] Transaction Performance Processing Council, "TPC Benchmarks," <http://www.tpc.org/>.
- [119] Y.-H. Tseng, C.-E. Huang, C. H. Kuo, Y. D. Chih, and C.-J. Lin, "High Density and Ultra Small Cell Size of Contact ReRAM (CR-RAM) in 90nm CMOS Logic Technology and Circuits," in *IEDM*, 2009.
- [120] H. Usui, L. Subramanian, K. Chang, and O. Mutlu, "SQUASH: Simple qos-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators," *arXiv CoRR*, 2015.

- [121] H. Usui, L. Subramanian, K. Chang, and O. Mutlu, "DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators," *ACM TACO*, vol. 12, no. 4, Jan. 2016.
- [122] H. Vandierendonck and A. Sez nec, "Fairness Metrics for Multi-threaded Processors," *IEEE CAL*, Feb 2011.
- [123] H. Wong *et al.*, "Phase change memory," *Proc. of the IEEE*, 2010.
- [124] D. Xiong, K. Huang, X. Jiang, and X. Yan, "Memory Access Scheduling Based on Dynamic Multilevel Priority in Shared DRAM Systems," *ACM TACO*, vol. 13, no. 4, Dec. 2016.
- [125] H. Yoon, J. Meza, R. Ausavarungnirun, R. Harding, and O. Mutlu, "Row Buffer Locality Aware Caching Policies for Hybrid Memories," in *ICCD*, 2012.
- [126] H. Yoon, J. Meza, N. Muralimanohar, N. P. Jouppi, and O. Mutlu, "Efficient Data Mapping and Buffering Techniques for Multilevel Cell Phase-change Memories," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 4, p. 40, 2014.
- [127] X. Yu, C. J. Hughes, N. Satish, O. Mutlu, and S. Devadas, "Banshee: Bandwidth-Efficient DRAM Caching via Software/Hardware Cooperation," in *MICRO*, 2017.
- [128] X. Yu, C. J. Hughes, N. Satish, O. Mutlu, and S. Devadas, "Banshee: Bandwidth-Efficient DRAM Caching via Software/Hardware Cooperation," arXiv:1704.02677 [CoRR], 2017.
- [129] W. Zhang *et al.*, "Exploring Phase Change Memory and 3D Die-Stacking for Power/Thermal Friendly, Fast and Durable Memory Architectures," in *PACT*, 2009.
- [130] J. Zhao, O. Mutlu, and Y. Xie, "FIRM: Fair and High-Performance Memory Control for Persistent Memory Systems," in *MICRO*, 2014.
- [131] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology," in *ISCA*, 2009.
- [132] W. K. Zuravleff and T. Robinson, "Controller for a Synchronous DRAM That Maximizes Throughput by Allowing Memory Requests and Commands to Be Issued Out of Order," US Patent No. 5,630,096, 1997.

Decoupling GPU Programming Models from Resource Management for Enhanced Programming Ease, Portability, and Performance

Nandita Vijaykumar¹ Kevin Hsieh¹ Gennady Pekhimenko^{2,3,1} Samira Khan⁴
Ashish Shrestha^{5,1} Saugata Ghose¹ Adwait Jog⁶ Phillip B. Gibbons¹ Onur Mutlu^{7,1}
¹Carnegie Mellon University ²University of Toronto ³Microsoft Research
⁴University of Virginia ⁵Intel ⁶College of William and Mary ⁷ETH Zürich

This paper summarizes the idea of Zorua, which was published in MICRO 2016 [88], and examines the work’s significance and future potential. The application resource specification—a static specification of several parameters such as the number of threads and the scratchpad memory usage per thread block—forms a critical component of modern GPU programming models. This specification determines the parallelism, and hence performance, of the application during execution because the corresponding on-chip hardware resources are allocated and managed based on this specification. This tight-coupling between the software-provided resource specification and resource management in hardware leads to significant challenges in programming ease, portability, and performance. Zorua is a new resource virtualization framework, that decouples the programmer-specified resource usage of a GPU application from the actual allocation in the on-chip hardware resources. Zorua enables this decoupling by virtualizing each resource transparently to the programmer.

The virtualization provided by Zorua builds on two key concepts—dynamic allocation of the on-chip resources, and their oversubscription using a swap space in memory. Zorua provides a holistic GPU resource virtualization strategy designed to (i) adaptively control the extent of oversubscription, and (ii) coordinate the dynamic management of multiple on-chip resources to maximize the effectiveness of virtualization. We demonstrate that by providing the illusion of more resources than physically available via controlled and coordinated virtualization, Zorua offers several important benefits: (i) **Programming Ease**. Zorua eases the burden on the programmer to provide code that is tuned to efficiently utilize the physically available on-chip resources. (ii) **Portability**. Zorua alleviates the necessity of re-tuning an application’s resource usage when porting the application across GPU generations. (iii) **Performance**. By dynamically allocating resources and carefully oversubscribing them when necessary, Zorua improves or retains the performance of applications that are already highly tuned to best utilize the resources. The holistic virtualization provided by Zorua has many other potential uses, e.g., fine-grained resource sharing among multiple kernels, low-latency preemption of GPU programs, and support for dynamic parallelism.

1. Motivation: Key Challenges in Modern GPUs

Modern Graphics Processing Units (GPUs) offer high performance and energy efficiency for many classes of applications by concurrently executing thousands of threads. In order to execute, each thread requires several major on-chip resources: (i) registers, (ii) scratchpad memory (if used in the program), and (iii) a thread slot in the thread scheduler that keeps all the bookkeeping information required for execution.

Today, these resources are *statically* allocated to threads based on several parameters—the number of threads per thread block, register usage per thread, and scratchpad usage per block. We refer to these static application parameters as the *resource specification* of the application. This resource specification forms a critical component of modern GPU programming models (e.g., CUDA [63], OpenCL [50]). The static allocation over a fixed set of hardware resources based on the software-specified resource specification creates a *tight coupling* between the program and the physical hardware resources. As a result of this tight coupling, for each application, there are only a few optimized resource specifications that maximize resource utilization. Picking a suboptimal specification leads to underutilization of resources and hence, very often, performance degradation. This leads to three key difficulties related to obtaining good performance on modern GPUs: programming ease, portability, and performance degradation.

Programming Ease. First, the burden falls upon the programmer to optimize the resource specification. For a naive programmer, this is a challenging task because, in addition to selecting a specification suited to an algorithm, the programmer needs to be aware of the details of the GPU architecture to fit the specification to the underlying hardware resources. This *tuning* is easy to get wrong because there are *many* highly suboptimal performance points in the specification space, and even a minor deviation from an optimized specification can lead to a drastic drop in performance due to lost parallelism. We refer to such drops as *performance cliffs*. Even a small change in one resource can result in a significant performance cliff, degrading performance by as much as 50%. Figure 1 depicts multiple sizable cliffs in an example application, when different resource specifications are used

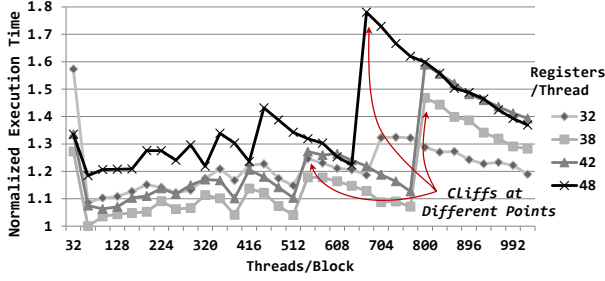


Figure 1: Performance cliffs in *Minimum Spanning Tree* (MST) when run on the NVIDIA GTX 745. Reproduced from [88].

when the program is run on a real modern GPU, the NVIDIA GTX 745.¹

Portability. Second, different GPUs have varying quantities of each of the resources. Hence, an optimized specification on one GPU may be highly suboptimal on another. This lack of *portability* necessitates that the programmer *re-tune* the resource specification of the application for *every* new GPU generation. This problem is especially significant in virtualized environments, such as data centers, cloud computing, or compute clusters, where the same program may run on a wide range of GPU architectures. Figure 2 depicts the 69% performance loss when porting optimized code from the NVIDIA Kepler [65]/Maxwell [66] architectures to the NVIDIA Fermi [64] architecture.

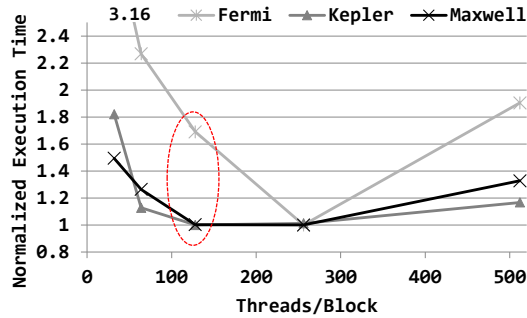


Figure 2: Performance variation across different GPU generations from NVIDIA (Fermi, Kepler, and Maxwell) for *Discrete Fourier Transform* (DCT). Reproduced from [88].

Performance. Third, for a programmer who chooses to employ software optimization tools (e.g., auto-tuners [21, 24, 49, 74, 75, 79]) or manually tailor the program to fit the hardware, performance is still constrained by the *fixed, static* resource specification. It is well known [27, 42, 48, 62, 87, 97] that the on-chip resource requirements of a GPU application vary throughout execution. Since the program (even after auto-tuning) has to *statically* specify its *worst-case* resource requirements, severe *dynamic underutilization* of several GPU resources ensues [87], leading to suboptimal performance.

¹Our MICRO 2016 paper [88] describes the experimental methodology for collecting these real system results.

2. A Holistic Approach to Resource Virtualization

To address these three challenges at the same time, we propose Zorua, a new framework that *decouples* an application’s resource specification from the available hardware resources by *virtualizing* all three major resources (i.e., scratchpad memory, register file, and thread slots) in a holistic manner. This virtualization provides the illusion of more resources to the GPU programmer and software than physically available, and enables the runtime system and the hardware to *dynamically* manage multiple resources in a manner that is transparent to the programmer.

2.1. Key Concepts

The virtualization strategy used by Zorua is built upon two key concepts. First, to mitigate performance cliffs when we do not have enough physical resources, we *oversubscribe* resources by a small amount at runtime, by leveraging their dynamic underutilization and maintaining a swap space (in main memory) for the extra resources required. Second, Zorua improves utilization by determining the runtime resource requirements of an application. It then allocates and deallocates resources dynamically, managing them (i) *independently* of each other to maximize each resource’s utilization; and (ii) in a *coordinated* manner, to enable efficient execution of each thread with all its required resources available.

Figure 3 depicts the high-level overview of the virtualization provided by Zorua. The *virtual space* refers to the *illusion* of the quantity of available resources. The *physical space* refers to the *actual* hardware resources (specific to the target GPU architecture), and the *swap space* refers to the resources that do *not* fit in the physical space and hence are *spilled* to other physical locations. For the register file and scratchpad memory, the swap space is mapped to the global memory space in the memory hierarchy. For threads, only those that are mapped to the physical space are available for scheduling and execution at any given time. If a thread is mapped to the swap space, its state (e.g., the PC) is saved in memory. Resources in the virtual space can be freely re-mapped between the physical and swap spaces to maintain the illusion of the virtual space resources.

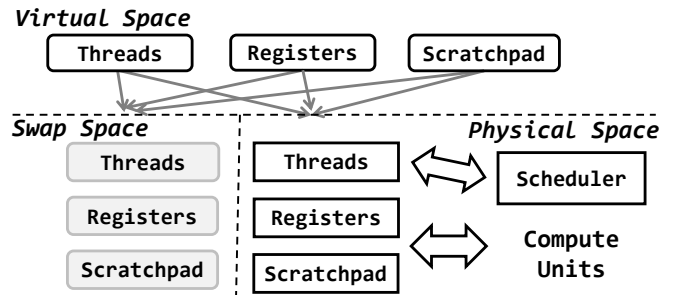


Figure 3: High-level overview of Zorua. Reproduced from [88].

2.2. Challenges in Virtualization

Unfortunately, oversubscription means that latency-critical resources, such as registers and scratchpad, may be swapped to memory at the time of access, resulting in high overheads in performance and energy. This leads to two critical challenges in designing a framework to enable virtualization. The first challenge is to effectively determine the *extent* of virtualization, i.e., by how much each resource appears to be larger than its real physical amount, such that we can *minimize* oversubscription while still reaping its benefits. This is difficult as the resource requirements continually vary during runtime. The second challenge is to minimize accesses to the swap space. This requires *coordination* in the virtualized management of *multiple resources*, so that enough of each resource is available on-chip at the same time when needed.

2.3. Design Ideas

To solve these challenges, Zorua employs two key ideas. First, we leverage the software (the compiler) to provide annotations with information regarding the future resource requirements of each *phase* of the application. This information enables the framework to make intelligent dynamic decisions ahead of time, with respect to both the extent of oversubscription and the allocation/deallocation of resources. Second, we use an adaptive runtime system to control the allocation of resources. This allows us to (i) dynamically alter the extent of oversubscription; and (ii) continuously coordinate the allocation of multiple on-chip resources and the mapping between their virtual and physical/swap spaces; depending on the varying runtime requirements of each thread. We briefly describe each design idea in turn.

2.3.1. Leveraging Software Annotations of Phase Characteristics. We observe that the runtime variation in resource requirements typically occurs at the granularity of *phases* of a few tens of instructions. This variation occurs because different parts of kernels perform different operations that require different resources. For example, loops that primarily load/store data from/to scratchpad memory tend to be less register heavy. Sections of code that perform specific computations (e.g., matrix transformation, graph manipulation), can either be register heavy or primarily operate out of scratchpad. Often, scratchpad memory is used for only short intervals [97], e.g., when data exchange between threads is required, such as for a reduction operation.

Figure 4 depicts a few example phases from the *N-Queens Solver (NQU)* [18] kernel. *NQU* is a scratchpad-heavy application, but it does not use the scratchpad at all during the initial computation phase. During its second phase, it performs its primary computation out of the scratchpad, using as much as 4224B. During its last phase, the scratchpad is used only for reducing results, which requires only 384B. There is also significant variation in the maximum number of live registers in the different phases, as shown in Figure 4.

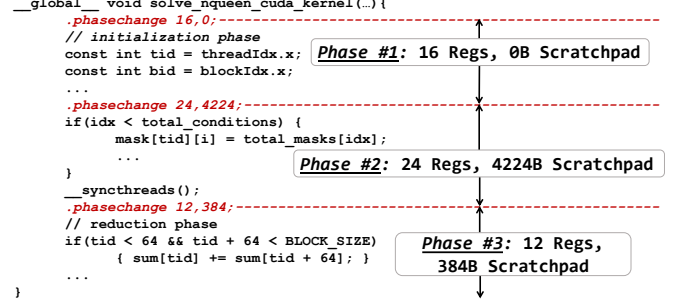


Figure 4: Example phases from *N-Queens Solver (NQU)*. Reproduced from [88].

In order to capture both the resource requirements as well as their variation over time, we partition the program into a number of *phases*. A phase is a sequence of instructions with sufficiently different resource requirements than adjacent phases.² Barrier or fence operations also indicate a change in requirements for a different reason—threads that are waiting at a barrier do not immediately require the thread slot that they are holding. We interpret barriers and fences as phase boundaries since they potentially alter the utilization of their thread slots. The compiler inserts special instructions called *phase specifiers* to mark the start of a new phase. Each phase specifier contains information regarding the resource requirements of the next phase. Phase changes are shown as “.phasechange” pragmas in Figure 4.

A phase forms the basic unit for resource allocation and deallocation, as well as for making oversubscription decisions. It offers a finer granularity than an *entire thread* to make such decisions. The phase specifiers provide information on the *future resource usage* of the thread at a phase boundary. This enables (i) preemptively controlling the extent of oversubscription at runtime, and (ii) dynamically allocating and deallocating resources at phase boundaries to maximize utilization of the physical resources.

2.3.2. Control with an Adaptive Runtime System. Phase specifiers provide information to make oversubscription and allocation/deallocation decisions. However, we still need a way to make decisions on the extent of oversubscription and appropriately allocate resources at runtime. To this end, we use an adaptive runtime system, which we refer to as the *coordinator*. Figure 5 presents an overview of the coordinator.

The virtual space enables the illusion of a larger amount of each of the resources than what is physically available, to adapt to different application requirements. This illusion enables higher thread-level parallelism than what can be achieved with solely the fixed, physically available resources, by allowing more threads to execute concurrently. The size of the virtual space at a given time determines this parallelism, and those threads that are effectively executed in parallel are referred to as *active threads*. All active threads have thread

²We refer the reader to Section 4.6 of our MICRO 2016 paper [88] for specific details on how phases are identified.

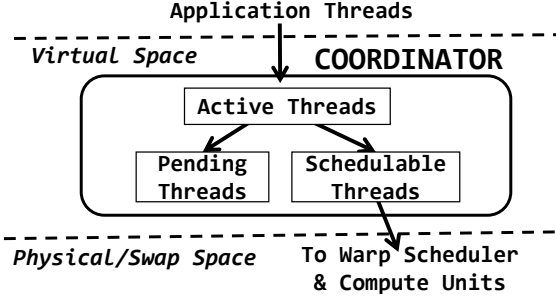


Figure 5: Overview of the coordinator. Reproduced from [88].

slots allocated to them in the virtual space (and hence can be executed), but some of them may *not* be mapped to the physical space at any given time. As discussed previously, the resource requirements of each application continuously change during execution. To adapt to these runtime changes, the coordinator leverages information from the phase specifiers to make decisions on oversubscription. The coordinator makes these decisions at every phase boundary and thereby controls the size of the virtual space for each resource.

2.4. Zorua: An Overview

To address the challenges in virtualization by leveraging the above ideas, Zorua employs a software-hardware code-sign that comprises three components: (i) **The compiler** annotates the program by adding special instructions (*phase specifiers*) to partition it into *phases* and to specify the resource needs of each phase of the application. (ii) **The coordinator**, a hardware-based adaptive runtime system, uses the compiler annotations to dynamically allocate/deallocate resources for each thread at phase boundaries. The coordinator plays the key role of continuously controlling the extent of the oversubscription at each phase boundary. (iii) **Hardware virtualization support** includes a mapping table for each resource to locate each virtual resource in either the physically available on-chip resources or the swap space in main memory, and the machinery to swap resources between the physical space and the swap space.

Zorua has two key hardware components: (i) the *coordinator* that contains queues to buffer the *pending threads* and control logic to make oversubscription and resource management decisions, and (ii) *resource mapping tables* to map each of the resources to their corresponding physical or swap spaces. Our MICRO 2016 paper [88] provides the detailed implementation of Zorua in Section 4. In particular, we describe several key issues, including how (1) Zorua determines the amount of oversubscription for each resource (Section 4.4 of [88]), (2) Zorua virtualizes each resource (Section 4.5 of [88]), and (3) the compiler identifies each phase (Section 4.6 of [88]).

3. Results

In this section, we evaluate the effectiveness of Zorua in improving programming ease, portability, and performance. Our

detailed experimental methodology is described in Section 5 of our MICRO 2016 paper [88]. More results are provided in Section 6 of [88].

3.1. Effect on Performance Variation and Cliffs

We first examine how Zorua alleviates the high variation in performance by reducing the impact of resource specifications on resource utilization. Figure 6 summarizes the range in performance across a wide range of resource specifications (indicating an undesirable dependence on the specification), for the baseline architecture, WLM (which allocates resources at the finer granularity of a warp [91]), and Zorua for a representative set of applications, using a Tukey box plot [61]. The boxes in the box plot represent the range between the first quartile (25%) and the third quartile (75%). The whiskers extending from the boxes represent the maximum and minimum points of the distribution, or $1.5 \times$ the length of the box, whichever is smaller. Any points that lie more than $1.5 \times$ the box length beyond the box are considered to be outliers [61], and are plotted as individual points. The line in the middle of the box represents the median, while the “X” represents the average. We make two major observations from Figure 6.

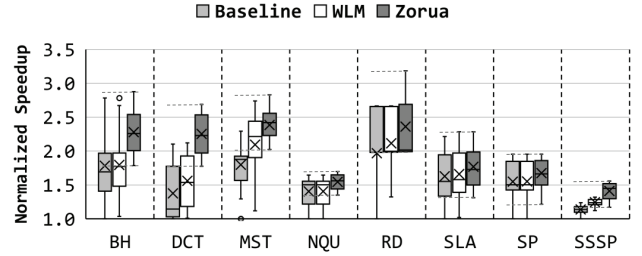


Figure 6: Normalized performance distribution. Reproduced from [88].

First, we find that Zorua significantly reduces the *performance range* across all evaluated resource specifications. Averaged across all of our applications, the worst resource specification for Baseline achieves 96.6% lower performance than the best performing resource specification. For WLM [91], this performance range reduces only slightly, to 88.3%. With Zorua, the performance range drops significantly, to 48.2%. We see drops in the performance range for *all* applications except SSSP. With SSSP, the range is already small to begin with (23.8% in Baseline), and Zorua exploits the dynamic underutilization, which improves performance but also adds a small amount of variation.

Second, while Zorua reduces the performance range, it also preserves or improves performance of the best performing points. As we examine in more detail in Section 3.2, the reduction in performance range occurs as a result of improved performance mainly at the lower end of the distribution.

To gain insight into how Zorua reduces the performance range and improves performance for the worst performing points, we analyze how it reduces performance cliffs. We study the tradeoff between resource specification and exe-

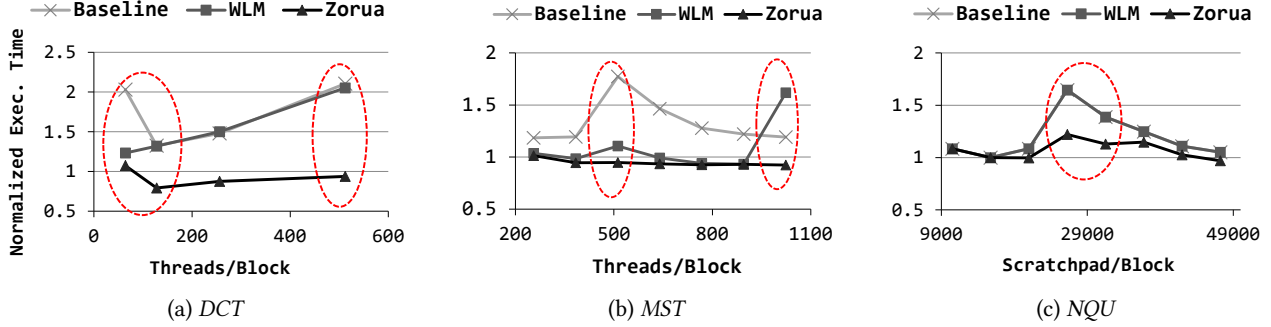


Figure 7: Effect on performance cliffs. Reproduced from [88].

cution time for three representative applications: *DCT* (Figure 7a), *MST* (Figure 7b), and *NQU* (Figure 7c). For all three figures, we normalize execution time to the *best* execution time under Baseline. We make two observations from the figures.

First, Zorua successfully mitigates the performance cliffs that occur in Baseline. For example, *DCT* and *MST* are both sensitive to the thread block size, as shown in Figures 7a and 7b, respectively. We have circled the locations at which cliffs exist in Baseline. Unlike Baseline, Zorua maintains more steady execution times across the number of threads per block, employing oversubscription to overcome the loss in parallelism due to insufficient on-chip resources. We see similar results across all of our applications.

Second, we observe that while WLM [91] can reduce some of the cliffs by mitigating the impact of large block sizes, many cliffs still exist under WLM (e.g., *NQU* in Figure 7c). This cliff in *NQU* occurs as a result of insufficient scratchpad memory, which cannot be handled by warp-level management. Similarly, the cliffs for *MST* (Figure 7b) also persist with WLM because *MST* has a lot of barrier operations, and the additional warps scheduled by WLM ultimately stall, waiting for other warps within the same block to acquire resources. We find that, with oversubscription, Zorua is able to smooth out those cliffs that WLM is unable to eliminate.

3.2. Effect on Performance

As Figure 6 shows, Zorua either retains or improves the best performing point for each application, compared to the Baseline. Zorua improves the best performing point for each application by 12.8% on average, and by as much as 27.8% (for *DCT*). This improvement comes from the improved parallelism obtained by exploiting the dynamic underutilization of resources, which exists *even for optimized specifications*. Applications such as *SP* and *SLA* have little dynamic underutilization, and hence do not show any performance improvement. *NQU* *does* have significant dynamic underutilization, but Zorua does not significantly improve the best performing point as the overhead of oversubscription outweighs the benefit, and Zorua dynamically chooses *not* to oversubscribe. We conclude that even for many specifications that are *opti-*

mized to fit the underlying hardware resources, Zorua is able to further improve performance.

We also note that, in addition to reducing performance variation and improving performance for optimized points, Zorua improves performance by 25.2% on average for all resource specifications across all evaluated applications.

3.3. Effect on Portability

Performance cliffs often behave differently across different GPU architectures, and can significantly shift the best performing resource specification point. We study how Zorua can ease the burden of performance tuning if an application has been already tuned for one GPU model, and is later ported to another GPU. To understand this, we define a new metric, *porting performance loss*, that quantifies the performance impact of porting an application without re-tuning it. To calculate this, we first normalize the execution time of each specification point to the execution time of the best performing specification point. We then pick a source GPU architecture (i.e., the architecture that the GPU was tuned for) and a target GPU architecture (i.e., the architecture that the code will run on), and find the point-to-point drop in performance (when the code is executed on the target GPU) for all points whose performance on the source GPU comes within 5% of the performance at the best performing specification point.³

Figure 8 shows the *maximum* porting performance loss for each application, across any two pairings of our three simulated GPU architectures (NVIDIA Fermi, Kepler, and Maxwell). We find that Zorua greatly reduces the maximum porting performance loss that occurs under both Baseline and WLM for all but one of our applications. On average, the maximum porting performance loss is 52.7% for Baseline, 51.0% for WLM, and only 23.9% for Zorua.

Notably, Zorua delivers significant improvements in portability for applications that previously suffered greatly when ported to another GPU, such as *DCT* and *MST*. For both of these applications, the performance variation differs so much

³We include any point within 5% of the best performance as there are often multiple points close to the best point, and the programmer may choose any of them.

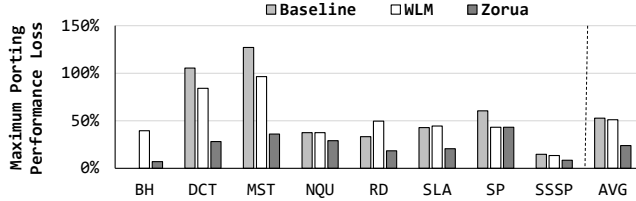


Figure 8: Maximum porting performance loss. Reproduced from [88].

between GPU architectures that, despite tuning the application on the source GPU to be within 5% of the best achievable performance, their performance on the target GPU is often more than twice as slow as the best achievable performance on the target platform. Zorua significantly lowers this porting performance loss down to 28.1% for *DCT* and 36.1% for *MST*. We also observe that for *BH*, Zorua actually slightly increases the porting performance loss with respect to the Baseline. This is because for Baseline, there are only two points that perform within the 5% margin for our metric, whereas with Zorua, we have five points that fall in that range. Despite this, the increase in porting performance loss for *BH* is low, deviating only 7.0% from the best performance.

We conclude that Zorua enhances portability of applications by reducing the impact of a change in the hardware resources for a given resource specification. For applications that have already been tuned on one platform, Zorua significantly lowers the penalty of not re-tuning for another platform, allowing programmers to save development time.

4. Related Work

To our knowledge, our MICRO 2016 paper [88] is the first work to propose a holistic framework to decouple a GPU application’s resource specification from its physical on-chip resource allocation by virtualizing multiple on-chip resources. This enables the illusion of more resources than what physically exists to the programmer, while the hardware resources are managed at runtime by employing a swap space (in main memory), transparently to the programmer. We design a new hardware/software cooperative framework to effectively virtualize multiple on-chip GPU resources in a controlled and coordinated manner, thus enabling many benefits of virtualization in GPUs.

We briefly discuss prior work related to different aspects of our proposal: (i) virtualization of resources, (ii) improving programming ease and portability, and (iii) more efficient management of on-chip resources.

Virtualization of Resources. *Virtualization* [20, 22, 33, 41] is a concept designed to provide the illusion, to the software and programmer, of more resources than what truly exists in physical hardware. It has been applied to the management of hardware resources in many different contexts [5, 10, 20, 22, 33, 41, 67, 89], with virtual memory [11, 22, 26, 41] being one of the oldest forms of virtualization that is commonly used in high-performance processors today. Ab-

straction of hardware resources and use of a level of indirection in their management leads to many benefits, including improved utilization, programmability, portability, isolation, protection, sharing, and oversubscription.

In this work, we apply the general principle of virtualization to the management of multiple on-chip resources in modern GPUs. Virtualization of on-chip resources offers the opportunity to alleviate many different challenges in modern GPUs. However, in this context, effectively adding a level of indirection introduces new challenges, necessitating the design of a new virtualization strategy. There are two key challenges. First, we need to dynamically determine the *extent* of the virtualization to reach an effective tradeoff between improved parallelism due to oversubscription and the latency/capacity overheads of swap space usage. Second, we need to coordinate the virtualization of *multiple* latency-critical on-chip resources. To our knowledge, this is the first work to propose a holistic software-hardware cooperative approach to virtualizing multiple on-chip resources in a controlled and coordinated manner that addresses these challenges, enabling the different benefits provided by virtualization in modern GPUs.

Prior works propose to virtualize a specific on-chip resource for specific benefits, mostly in the CPU context. For example, in CPUs, the concept of virtualized registers was first used in the IBM 360 [5] and DEC PDP-10 [10] architectures to allow logical registers to be mapped to either fast yet expensive physical registers, or slow and cheap memory. More recent works [67, 93, 94], propose to virtualize registers to increase the effective register file size to much larger register counts. This increases the number of thread contexts that can be supported in a multi-threaded processor [67], or reduces register spills and fills [93, 94]. Other works propose to virtualize on-chip resources in CPUs (e.g., [15, 19, 25, 31, 99]). In GPUs, Jeon et al. [42] propose to virtualize the register file by dynamically allocating and deallocating physical registers to enable more parallelism with smaller, more power-efficient physical register files. Concurrent to this work, Yoon et al. [98] propose an approach to virtualize thread slots to increase thread-level parallelism. These works propose specific virtualization mechanisms for a single resource for specific benefits. None of these works provide a cohesive virtualization mechanism for *multiple* on-chip GPU resources in a controlled and coordinated manner, which forms a key contribution of our MICRO 2016 work.

Enhancing Programming Ease and Portability. There is a large body of work that aims to improve programmability and portability of modern GPU applications using software tools, such as auto-tuners [21, 24, 49, 74, 75, 79], optimizing compilers [17, 37, 47, 59, 95, 96], and high-level programming languages and runtimes [23, 35, 72, 85]. These tools tackle a multitude of optimization challenges, and have been demonstrated to be very effective in generating high-performance portable code. They can also be used to tune the resource

specification. However, there are several shortcomings in these approaches. First, these tools often require profiling runs [17, 21, 75, 79, 95, 96] on the GPU to determine the best performing resource specifications. These runs have to be repeated for each new input set and GPU generation. Second, software-based approaches still require significant programmer effort to write code in a manner that can be exploited by these approaches to optimize the resource specifications. Third, selecting the best performing resource specifications statically using software tools is a challenging task in virtualized environments (e.g., cloud computing, data centers), where it is unclear which kernels may be run together on the same SM or where it is not known, a priori, which GPU generation the application may execute on. Finally, software tools assume a fixed amount of available resources. This leads to runtime underutilization due to static allocation of resources, which cannot be addressed by these tools.

In contrast, the programmability and portability benefits provided by Zorua require no programmer effort in optimizing resource specifications. Furthermore, these auto-tuners and compilers can be used in conjunction with Zorua to further improve performance.

Efficient Resource Management. Prior works aim to improve parallelism by increasing resource utilization using hardware-based [6, 7, 30, 42, 45, 46, 55, 57, 62, 71, 84, 86, 91, 97], software-based [32, 36, 53, 58, 68, 92, 97], and hardware-software cooperative [8, 9, 43, 44, 73, 81, 82, 87] approaches. Among these works, the closest to ours are [42, 98] (discussed earlier), [97], and [91]. These approaches propose efficient techniques to dynamically manage a single resource, and can be used along with Zorua to improve resource efficiency further. Yang et al. [97] aim to maximize utilization of the scratchpad with software techniques, and by dynamically allocating/deallocating scratchpad memory. Xiang et al. [91] propose to improve resource utilization by scheduling threads at the finer granularity of a warp rather than a thread block. This approach can help alleviate performance cliffs, but not in the presence of synchronization or scratchpad memory, nor does it address the dynamic underutilization within a thread during runtime. We quantitatively compare to this approach in Section 3 and demonstrate Zorua’s benefits over it.

Other works leverage resource underutilization to improve energy efficiency [2, 27, 28, 29, 42] or perform other useful work [54, 87]. These works are complementary to Zorua.

5. Significance and Long-Term Impact

In this section, we describe the significance and long-term impact of our MICRO 2016 work, Zorua, by delineating its novelty, what it can enable in future systems, and new research directions that it triggers.

5.1. Novelty

- This is the first work that takes a holistic approach to decoupling a GPU application’s resource specification from its

physical on-chip resource allocation via the use of virtualization. We develop a comprehensive virtualization framework that provides *controlled* and *coordinated* virtualization of *multiple* on-chip resources to maximize the effectiveness of virtualization.

- Making GPUs easy to program is critical for their widespread use, and also to achieve the high performance promised by the massively parallel architecture. A key limiting factor in GPU programming today is the burden placed on the programmer in finding a hardware resource specification that achieves very high performance. This is the first work to ease that burden without compromising performance by virtualizing the major hardware resources programmers are required to manage today.

- Portability across GPU architectures is vital in environments such as cloud computing and data centers to achieve predictably good performance, *irrespective* of the GPU generation the application is executing on. This is the first work to tackle the portability challenges that arise from the programmer’s management of the fixed on-chip resources with a holistic resource virtualization strategy.

5.2. What Zorua Can Enable in Future Systems

GPUs have emerged as the dominant massively parallel GPU architecture, used as the platform of choice for a wide range of parallel applications from machine learning to scientific simulation. However, there are a number of key challenges that limit the adoption of GPUs across broader classes of applications and environments, e.g., data centers, cloud computing, etc. Programmability and portability of GPU applications are two such challenges. But future GPUs will need to address several other challenges before truly becoming first-class compute engines. As we describe below, we believe that our work can help address some of these other challenges.

Multiprogramming in Virtualized Environments. Zorua lends itself to easily addressing two key challenges in enabling multiprogramming in virtualized environments today:

Fine-grained resource sharing across kernels: Zorua manages the different resources independently and at a fine granularity, using a dynamic runtime system. Hence, Zorua can be extended to support fine-grained sharing and partitioning of resources across multiple kernels to enable efficient multiprogramming in GPUs. Zorua enables better resource utilization in these multiprogrammed environments, while providing the ability to control the partitioning of resources at runtime to provide QoS, fairness, etc., by leveraging the hardware runtime system. Zorua can work synergistically with systems such as Mosaic [8] and MASK [9], which enable efficient memory virtualization techniques for GPUs, to enable true full-system multi-kernel execution.

Preemptive multitasking: Another key challenge in enabling true multiprogramming in GPUs is enabling rapid preemption of kernels [69,83,90]. Context switching on GPUs incurs a very high latency and overhead, as a result of the large amount of register file/scratchpad state that needs to be saved before a new kernel can be executed. Zorua enables fine-grained management and virtualization of on-chip resources. It can be naturally extended to enable quick preemption of a task via intelligent management of the swap space and the mapping tables. It can also work synergistically with CABA [87], framework for assist warp execution in GPUs, to provide flexible and efficient support for multitasking and context switching.

Support for Other Parallel Programming Paradigms. The fixed static resource allocation for each thread in modern GPU architectures requires statically dictating the parallelism for the program throughout its execution. Other forms of parallel execution that are *dynamic* (e.g., CILK [12]) require more flexible allocation of resources at runtime, and are hence more challenging to enable on GPUs. Examples of this include *nested parallelism* [56], where a kernel can dynamically spawn new kernels or thread blocks, and *helper threads* [87] to utilize idle resource at runtime to perform different optimizations or background tasks in parallel. Zorua makes it easy to enable these paradigms by providing on-demand dynamic allocation of resources.

Energy Efficiency, Scalability, and Reliability. To support massive parallelism, on-chip resources are a precious and critical resource. However, these resources *cannot* grow arbitrarily large as GPUs continue to be area-limited and on-chip memory tends to be extremely power hungry and area intensive [2, 27, 28, 42, 73, 98], which are trends we believe will become increasingly important for the foreseeable future. Furthermore, complex thread schedulers that can select a thread for execution from an increasingly large thread pool are required. Zorua enables using smaller register files, scratchpad memory and less complex or fewer thread schedulers to save power and area while still retaining or improving parallelism. The indirection offered by Zorua, along with the dynamic management of resources, could also enable better reliability. The virtualization framework trivially allows portions of a resource that contain hard or soft faults to be remapped to other portions of the resource that do not contain faults, or to spare structures, thereby increasing the error tolerance of these resources.

5.3. New Research Directions Zorua Enables

Zorua opens up several new avenues for more research, which we briefly discuss here.

Flexible Programming Models for GPUs and Heterogeneous Systems. By providing a flexible but dynamically controlled view of the on-chip hardware resources, Zorua changes the abstraction of the on-chip resources that is offered to the programmer and software. This offers the op-

portunity to rethink resource management in GPUs from the ground up. One could envision more powerful resource allocation and better programmability with programming models that do *not* require static resource specification, leaving the compiler/runtime system and the underlying virtualized framework to *completely* handle *all* forms of on-chip resource allocation, unconstrained by the fixed physical resources in a specific GPU, entirely at runtime. This is especially significant in future systems that are likely to support a wide range of compute engines and accelerators, making it important to be able to write high-level code that can be partitioned easily, efficiently, and at a fine granularity across any set of accelerators, without statically tuning any code segment to run efficiently on the GPU.

Virtualization-Aware Compilation and Auto-Tuning. Zorua changes the contract between the hardware and software to provide a more powerful resource abstraction (in the software) that is *flexible and dynamic*, by pushing some more functionality to the hardware, which can more easily react to runtime resource requirements of the program. We can re-imagine compilers and auto-tuners to be more intelligent, leveraging this new abstraction and, hence the virtualization, to deliver more efficient and high-performing code optimizations that are *not* possible with the *fixed* and *static* abstractions of today. They could, for example, *leverage* the oversubscription and dynamic management that Zorua provides to tune the code to more aggressively use resources.

Support for System-Level Tasks on GPUs. As GPUs become increasingly general purpose, a key requirement is better integration with the CPU operating system, and with complex distributed software systems such as those employed for large-scale distributed machine learning [1, 39] or graph processing [3, 4, 60]. If GPUs are architected to be first-class compute engines, rather than the slave devices they are today, they can be programmed and utilized in the same manner as a modern CPU. This integration requires the GPU execution model to support system-level tasks like interrupts, exceptions, etc. and more generally provide support for access to distributed file systems, disk I/O, or network communication. Support for these tasks and execution models require dynamic provisioning of resources for execution of system-level code. Zorua provides a building block to enable this.

Applicability to General Resource Management in Accelerators. Zorua uses a program *phase* as the granularity for managing resources. This allows handling resources across phases *dynamically*, while leveraging *static* information regarding resource requirements from the software by inserting annotations at phase boundaries. Future work could potentially investigate the applicability of the same approach to manage resources and parallelism in *other* accelerators (e.g., processing-in-memory accelerators [3, 4, 13, 14, 34, 38, 40, 51, 52, 70, 77, 78, 80, 100] or direct-memory access engines [16, 55, 76]) that require efficient dy-

dynamic management of large amounts of particular critical resources.

6. Conclusion

We propose Zorua, a new framework that decouples the application resource specification from the allocation in the physical hardware resources (i.e., registers, scratchpad memory, and thread slots) in GPUs. Zorua encompasses a holistic virtualization strategy to effectively virtualize multiple latency-critical on-chip resources in a controlled and coordinated manner. We demonstrate that by providing the illusion of more resources than physically available, via dynamic management of resources and the judicious use of a swap space in main memory, Zorua enhances (i) *programming ease* (by reducing the performance penalty of suboptimal resource specification), (ii) *portability* (by reducing the impact of different hardware configurations), and (iii) *performance* for code with an optimized resource specification (by leveraging dynamic underutilization of resources). We conclude that Zorua is an effective, holistic virtualization framework for GPUs. We believe that the indirection provided by Zorua’s virtualization mechanism makes it a generic framework that can address other challenges in modern GPUs. For example, Zorua can enable fine-grained resource sharing and partitioning among multiple kernels/applications, as well as low-latency preemption of GPU programs. We hope that future work explores these promising directions, building on the insights and the framework developed in our MICRO 2016 paper.

Acknowledgments

We thank the reviewers and our shepherd for their valuable suggestions. We thank the members of the SAFARI group for their feedback and the stimulating research environment they provide. Special thanks to Vivek Seshadri, Kathryn McKinley, Steve Keckler, Evgeny Bolotin, and Mike O’Connor for their feedback during various stages of this project. We acknowledge the support of our industrial partners: Facebook, Google, IBM, Intel, Microsoft, NVIDIA, Qualcomm, Samsung, and VMware. This research was partially supported by NSF (grant 1409723), the Intel Science and Technology Center for Cloud Computing, and the Semiconductor Research Corporation.

References

- [1] M. Abadi *et al.*, “TensorFlow: A System for Large-Scale Machine Learning,” in *OSDI*, 2016.
- [2] M. Abdel-Majeed *et al.*, “Warped Register File: A Power Efficient Register File for GPGPUs,” in *HPCA*, 2013.
- [3] J. Ahn *et al.*, “PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture,” in *ISCA*, 2015.
- [4] J. Ahn *et al.*, “A scalable processing-in-memory accelerator for parallel graph processing,” in *ISCA*, 2015.
- [5] G. M. Amdahl *et al.*, “Architecture of the IBM System/360,” *IBM JRD*, 1964.
- [6] R. Ausavarungnirun *et al.*, “Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance,” *PACT*, 2015.
- [7] R. Ausavarungnirun *et al.*, “Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems,” in *ISCA*, 2012.
- [8] R. Ausavarungnirun *et al.*, “Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes,” in *MICRO*, 2017.
- [9] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. J. Rossbach, and O. Mutlu, “MASK: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency,” in *ASPLOS*, 2018.
- [10] C. G. Bell *et al.*, “The Evolution of the DEC System 10,” *CACM*, 1978.
- [11] A. Bensoussan *et al.*, “The Multics Virtual Memory,” in *SOSP*, 1969.
- [12] R. D. Blumofe *et al.*, “Cilk: An efficient multithreaded runtime system,” in *ASPLOS*, 1995.
- [13] A. Boroumand *et al.*, “Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks,” in *ASPLOS*, 2018.
- [14] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, K. Hsieh, K. T. Malladi, H. Zheng, and O. Mutlu, “LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory,” *CAL*, 2016.
- [15] E. Brekelbaum *et al.*, “Hierarchical scheduling windows,” in *MICRO*, 2002.
- [16] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu, “Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM,” in *HPCA*, 2016.
- [17] G. Chen *et al.*, “PORPLE: An extensible optimizer for portable data placement on GPU,” in *MICRO*, 2014.
- [18] P. Chen, “N-Queens solver,” <http://forums.nvidia.com/index.php?showtopic=76893>, 2008.
- [19] H. Cook *et al.*, “Virtual local stores: Enabling software-managed memory hierarchies in mainstream computing environments,” Univ. of California, Berkeley, EECS Dept., Tech. Rep. UCB/EECS-2009-131, 2009.
- [20] R. J. Creasy, “The Origin of the VM/370 Time-sharing System,” *IBM JRD*, 1981.
- [21] A. Davidson *et al.*, “Toward Techniques for Auto-Tuning GPU Algorithms,” in *Applied Parallel and Scientific Computing*. Springer, 2010.
- [22] P. J. Denning, “Virtual memory,” *ACM Comput. Surv.*, 1970.
- [23] R. Dolbeau *et al.*, “HMPP: A hybrid multi-core parallel programming environment,” in *GPGPU*, 2007.
- [24] Y. Dotsenko *et al.*, “Auto-tuning of Fast Fourier Transform on Graphics Processors,” *PPoPP*, 2011.
- [25] M. Erez *et al.*, “Spills, Fills, and Kills - An Architecture for Reducing Register-Memory Traffic,” Stanford Univ., Concurrent VLSI Architecture Group, Tech. Rep. TR-23, July 2000.
- [26] J. Fotheringham, “Dynamic storage allocation in the Atlas computer, including an automatic use of a backing store,” *CACM*, 1961.
- [27] M. Gebhart *et al.*, “A Compile-time Managed Multi-level Register File Hierarchy,” in *MICRO*, 2011.
- [28] M. Gebhart *et al.*, “Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors,” *ISCA*, 2011.
- [29] M. Gebhart *et al.*, “A hierarchical thread scheduler and register file for energy-efficient throughput processors,” *TOCS*, 2012.
- [30] M. Gebhart *et al.*, “Unifying Primary Cache, Scratch, and Register File Memories in a Throughput Processor,” in *MICRO*, 2012.
- [31] A. Gonzalez *et al.*, “Virtual-physical registers,” in *HPCA*, 1998.
- [32] C. Gregg *et al.*, “Fine-grained resource sharing for concurrent GPGPU kernels,” in *HotPar*, 2012.
- [33] P. H. Gum, “System/370 Extended Architecture: Facilities for Virtual Machines,” *IBM JRD*, 1983.
- [34] Q. Guo, N. Alachiotis, B. Akin, F. Sadi, G. Xu, T.-M. Low, L. Pileggi, J. C. Hoe, and F. Franchetti, “3D-Stacked Memory-Side Acceleration: Accelerator and System Design,” in *WandP*, 2014.
- [35] T. D. Han *et al.*, “hiCUDA: High-Level GPGPU Programming,” *TPDS*, 2011.
- [36] A. B. Hayes *et al.*, “Unified On-chip Memory Allocation for SIMT Architecture,” in *ICS*, 2014.
- [37] A. H. Hormati *et al.*, “Sponge: Portable Stream Programming on Graphics Engines,” *ASPLOS*, 2011.
- [38] K. Hsieh *et al.*, “Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation,” in *ICCD*, 2016.
- [39] K. Hsieh *et al.*, “Gaia: Geo-Distributed Machine Learning Approaching LAN Speeds,” in *NSDI*, 2016.
- [40] K. Hsieh *et al.*, “Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems,” in *ISCA*, 2016.
- [41] B. Jacob *et al.*, “Virtual memory in contemporary microprocessors,” *IEEE Micro*, 1998.
- [42] H. Jeon *et al.*, “GPU register file virtualization,” in *MICRO*, 2015.
- [43] J. A. Joao *et al.*, “Bottleneck identification and scheduling in multithreaded applications,” in *ASPLOS*, 2012.
- [44] J. A. Joao *et al.*, “Utility-Based Acceleration of Multithreaded Applications on Asymmetric CMPs,” in *ISCA*, 2013.
- [45] A. Jog *et al.*, “Orchestrated Scheduling and Prefetching for GPGPUs,” in *ISCA*, 2013.
- [46] A. Jog *et al.*, “OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance,” in *ASPLOS*, 2013.
- [47] J. C. Juega *et al.*, “Adaptive Mapping and Parameter Selection Scheme to Improve Automatic Code Generation for GPUs,” in *CGO*, 2014.
- [48] O. Kayiran *et al.*, “ μ C-States: Fine-grained GPU Datapath Power Management,” in *PACT*, 2016.
- [49] M. Khan *et al.*, “A Script-based Autotuning Compiler System to Generate High-performance CUDA Code,” *TACO*, 2013.
- [50] Khronos Group, “OpenCL,” <https://www.khronos.org/opencl/>.

- [51] J. S. Kim, D. Senol, H. Xin, D. Lee, S. Ghose, M. Alser, H. Hassan, O. Ergin, C. Alkan, and O. Mutlu, "GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-in-Memory Technologies," *BMC Genomics*, 2018.
- [52] P. M. Kogge, "EXECUBE—A New Architecture for Scaleable MPPs," in *ICPP*, 1994.
- [53] R. Komuravelli *et al.*, "Stash: Have your scratchpad and cache it too," in *ISCA*, 2015.
- [54] N. B. Lakshminarayana *et al.*, "Spare register aware prefetching for graph algorithms on GPUs," in *HPCA*, 2014.
- [55] D. Lee *et al.*, "Decoupled direct memory access: Isolating CPU and IO traffic by leveraging a dual-data-port DRAM," in *PACT*, 2015.
- [56] H. Lee *et al.*, "Locality-aware mapping of nested parallel patterns on GPUs," in *MICRO*, 2014.
- [57] M. Lee *et al.*, "Improving GPGPU resource utilization through alternative thread block scheduling," in *HPCA*, 2014.
- [58] C. Li *et al.*, "Automatic data placement into GPU on-chip memory resources," in *CGO*, 2015.
- [59] Y. Liu *et al.*, "A cross-input adaptive framework for GPU program optimizations," in *IPDPS*, 2009.
- [60] Y. Low *et al.*, "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud," *Proc. VLDB Endow.*, 2012.
- [61] R. McGill *et al.*, "Variations of box plots," *The American Statistician*, 1978.
- [62] V. Narasiman *et al.*, "Improving GPU Performance via Large Warps and Two-level Warp Scheduling," in *MICRO*, 2011.
- [63] NVIDIA, "CUDA," <https://developer.nvidia.com/about-cuda>.
- [64] NVIDIA, "Fermi: NVIDIA's Next Generation CUDA Compute Architecture," 2011.
- [65] NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," 2012.
- [66] NVIDIA, "Maxwell: NVIDIA's Next Generation CUDA Compute Architecture," <https://developer.nvidia.com/maxwell-compute-architecture>, 2014.
- [67] D. W. Oehmke *et al.*, "How to Fake 1000 Registers," in *MICRO*, 2005.
- [68] S. Pai *et al.*, "Improving GPGPU concurrency with elastic kernels," in *ASPLOS*, 2013.
- [69] J. Park *et al.*, "Chimera: Collaborative Preemption for Multitasking on a Shared GPU," *ASPLOS*, 2015.
- [70] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A Case for Intelligent RAM," *IEEE Micro*, 1997.
- [71] G. Pekhimenko *et al.*, "Toggle-aware compression for GPUs," in *HPCA*, 2016.
- [72] J. Ragan-Kelley *et al.*, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *PLDI*, 2013.
- [73] M. Sadrosadati *et al.*, "LTRF: Enabling High-Capacity Register Files for GPUs via Hardware/Software Cooperative Register Prefetching," in *ASPLOS*, 2018.
- [74] K. Sato *et al.*, *Automatic Tuning of CUDA Execution Parameters for Stencil Processing*. New York, NY: Springer-Verlag, 2010.
- [75] C. A. Schaefer *et al.*, "Atune-IL: An instrumentation language for auto-tuning parallel applications," in *Euro-Par*, 2009.
- [76] V. Seshadri *et al.*, "RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization," in *MICRO*, 2013.
- [77] V. Seshadri *et al.*, "Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology," in *MICRO*, 2017.
- [78] D. E. Shaw, S. J. Stolfo, H. Ibrahim, B. Hillyer, G. Wiederhold, and J. Andrews, "The NON-VON Database Machine: A Brief Overview," *IEEE Database Eng. Bull.*, 1981.
- [79] K. Spafford *et al.*, "Maestro: data orchestration and tuning for OpenCL devices," in *Euro-Par*, 2010.
- [80] H. S. Stone, "A Logic-in-Memory Computer," *IEEE TC*, 1970.
- [81] M. A. Suleman *et al.*, "Data marshaling for multi-core architectures," in *ISCA*, 2010.
- [82] M. A. Suleman *et al.*, "Accelerating critical section execution with asymmetric multi-core architectures," in *ASPLOS*, 2009.
- [83] I. Tanasic *et al.*, "Enabling Preemptive Multiprogramming on GPUs," in *ISCA*, 2014.
- [84] D. Tarjan *et al.*, "On demand register allocation and deallocation for a multithreaded processor," US Patent Application 20110161616, 2011.
- [85] S.-Z. Ueng *et al.*, "CUDA-Lite: Reducing GPU Programming Complexity," in *LCPC*, 2008.
- [86] H. Usui *et al.*, "DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators," *TACO*, 2016.
- [87] N. Vijaykumar *et al.*, "A Case for Core-Assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps," in *ISCA*, 2015.
- [88] N. Vijaykumar *et al.*, "Zorua: A Holistic Approach to Resource Virtualization in GPUs," in *MICRO*, 2016.
- [89] C. A. Waldspurger, "Memory Resource Management in VMware ESX Server," in *OSDI*, 2002.
- [90] Z. Wang *et al.*, "Simultaneous Multikernel GPU: Multi-tasking Throughput Processors via Fine-Grained Sharing," in *HPCA*, 2016.
- [91] P. Xiang *et al.*, "Warp-level divergence in GPUs: Characterization, impact, and mitigation," in *HPCA*, 2014.
- [92] X. Xie *et al.*, "Enabling coordinated register allocation and thread-level parallelism optimization for GPUs," in *MICRO*, 2015.
- [93] J. Yan *et al.*, "Virtual Registers: Reducing Register Pressure Without Enlarging the Register File," in *HIPEAC*, 2007.
- [94] J. Yan *et al.*, "Exploiting Virtual Registers to Reduce Pressure on Real Registers," *TACO*, 2008.
- [95] Y. Yang *et al.*, "A GPGPU Compiler for Memory Optimization and Parallelism Management," *PLDI*, 2010.
- [96] Y. Yang *et al.*, "A Unified Optimizing Compiler Framework for Different GPGPU Architectures," *TACO*, 2012.
- [97] Y. Yang *et al.*, "Shared memory multiplexing: a novel way to improve GPGPU throughput," in *PACT*, 2012.
- [98] M. Yoon *et al.*, "Virtual Thread: Maximizing Thread-Level Parallelism beyond GPU Scheduling Limit," in *ISCA*, 2016.
- [99] J. Zalamea *et al.*, "Two-level Hierarchical Register File Organization for VLIW Processors," in *MICRO*, 2000.
- [100] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, "TOP-PIM: Throughput-oriented Programmable Processing in Memory," in *HPDC*, 2014.

Holistic Management of the GPGPU Memory Hierarchy to Manage Warp-level Latency Tolerance

Rachata Ausavarungnirun¹ Saugata Ghose¹ Onur Kayiran^{2,3}
Gabriel H. Loh² Chita R. Das³ Mahmut T. Kandemir³ Onur Mutlu^{4,1}

¹Carnegie Mellon University ²AMD Research ³Pennsylvania State University ⁴ETH Zürich

This paper summarizes the idea of Memory Divergence Correction (MeDiC), which was published at PACT 2015 [6], and examines the work’s significance and future potential. In a modern GPU architecture, all threads within a warp execute the same instruction in lockstep. For a memory instruction, this can lead to memory divergence: the memory requests for some threads are serviced early, while the remaining requests incur long latencies. This divergence stalls the warp, as it cannot execute the next instruction until all requests from the current instruction complete.

In this work, we make three new observations. First, GPGPU warps exhibit heterogeneous memory divergence behavior at the shared cache: some warps have most of their requests hit in the cache (high cache utility), while other warps see most of their request miss (low cache utility). Second, a warp retains the same divergence behavior for long periods of execution. Third, due to high memory level parallelism, requests going to the shared cache can incur queuing delays as large as hundreds of cycles, exacerbating the effects of memory divergence.

We propose a set of techniques, collectively called Memory Divergence Correction (MeDiC), that reduce the negative performance impact of memory divergence and cache queuing. MeDiC uses online warp divergence characterization to guide three components: (1) a cache bypassing mechanism that exploits the latency tolerance of low cache utility warps to both alleviate queuing delay and increase the hit rate for high cache utility warps, (2) a cache insertion policy that prevents data from high cache utility warps from being prematurely evicted, and (3) a memory controller that prioritizes the few requests received from high cache utility warps to minimize stall time. We compare MeDiC to four cache management techniques, and find that it delivers an average speedup of 21.8%, and 20.1% higher energy efficiency, over a state-of-the-art GPU cache management mechanism across 15 different GPGPU applications.

1. Introduction

Graphics Processing Units (GPUs) have enormous parallel processing power to leverage thread-level parallelism. GPU applications are usually broken down into thousands of threads, allowing GPUs to use *fine-grained multithreading* [128, 136] to prevent GPU cores from stalling due to dependencies and long memory latencies. Ideally, there should always be available threads for GPU cores to continue execution, preventing stalls within the core. GPUs also take advantage of the *SIMD* (Single Instruction, Multiple Data)

execution model [30]. The thousands of threads within a GPU application are clustered into *thread blocks*, each of which contains multiple smaller bundles (*warps*) of threads that run concurrently. Each thread in a warp executes the same instruction on a different piece of data. A warp completes an instruction when all threads in the warp complete the instruction.

While many GPGPU applications can tolerate a significant amount of memory latency due to their parallelism and the use of fine-grained multithreading, *memory divergence* (where the threads of a warp reach a memory instruction, and some of the threads’ memory requests take longer to service than others) can significantly increase the stall time of a warp [51, 52, 63, 75, 89, 101, 116, 117, 155]. Because all threads within a warp operate in lockstep due to the SIMD execution model, the warp cannot proceed to the next instruction until the *slowest* request within the warp completes. Figures 1a and 1b show examples of memory divergence within a warp. Figure 1a shows a *mostly-hit warp*, where most of the warp’s memory accesses hit in the cache (❶). Only a single access misses in the cache and must go to main memory (❷). As a result, the *entire warp* is stalled until the much longer cache miss completes. Figure 1b shows a *mostly-miss warp*, where most of the warp’s memory requests miss in the cache (❸), resulting in many accesses to main memory. Even though some requests are cache hits (❹), these do not benefit the execution time of the warp since the execution of the warp ends when the slowest thread finishes the instruction.

Based on these three observations, we aim to devise a mechanism that has two major goals: (1) convert mostly-hit warps into *all-hit warps* (warps where *all* requests hit in the cache, as shown in Figure 1c), and (2) convert mostly-miss warps into *all-miss warps* (warps where *none* of the requests hit in the cache, as shown in Figure 1d). As we can see in Figure 1a, the stall time due to memory divergence for the mostly-hit warp can be eliminated by converting only the single cache miss (❷) into a hit. Doing so requires additional cache space. If we convert the two cache hits of the mostly-miss warp (Figure 1b, ❹) into cache misses, we can allocate the cache space previously used by these hits to the mostly-hit warp, thus converting the mostly-hit warp into an all-hit warp. Though the mostly-miss warp is now an all-miss warp (Figure 1d), it incurs no extra stall penalty, as the warp was already waiting on the other six cache misses to complete.

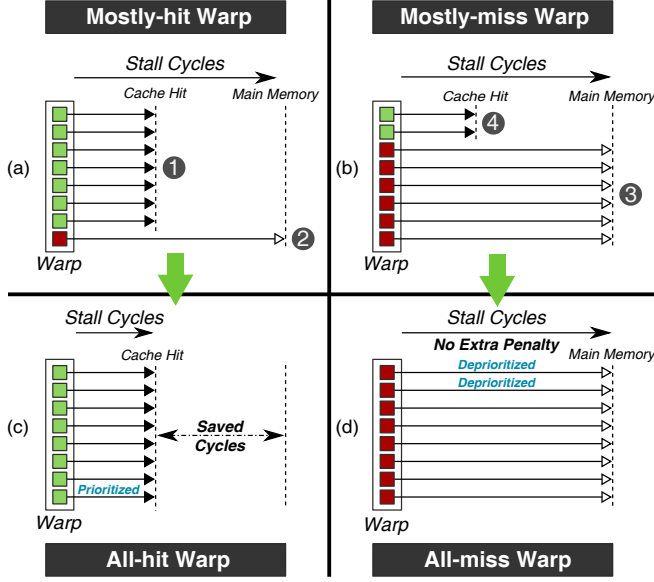


Figure 1: Memory divergence within a warp. (a) and (b) show the heterogeneity between *mostly-hit* and *mostly-miss* warps, respectively. (c) and (d) show the change in stall time from converting *mostly-hit* warps into *all-hit* warps, and *mostly-miss* warps into *all-miss* warps, respectively. Reproduced from [6].

Moreover, now that it is an all-miss warp, we can predict that its future memory requests will also not be in the L2 cache. Based on this prediction, we can simply have these requests *bypass the cache*. By doing so, the requests from the all-miss warp can completely avoid unnecessary L2 access and queuing delays, and enable the use of L2 cache bandwidth and buffer space by warps that benefit from the L2 cache. This decreases the total number of requests going to the L2 cache, thus reducing the queuing latencies for requests from *mostly-hit* and *all-hit* warps, as there is less contention.

2. Observation on GPU Memory Divergence

We make three new key observations about memory divergence (at the shared L2 cache). First, we observe that the degree of memory divergence can differ across warps (as illustrated in Figure 1). This inter-warp heterogeneity affects how well each warp takes advantage of the shared cache. Second, we observe that a warp’s memory divergence behavior tends to remain stable for long periods of execution, making it predictable. Third, we observe that requests to the shared cache experience long queuing delays due to the large amount of parallelism in GPGPU programs, which exacerbates the memory divergence problem and slows down GPU execution. Next, we describe each of these observations in detail and motivate our solutions.

2.1. Memory Divergence Heterogeneity

There is *heterogeneity across warps* in the degree of memory divergence experienced by each warp at the shared L2 cache. Figures 1a and 1b show examples of two different *types of warps* that exhibit different degrees of memory divergence.

We observe that different warps have different amounts of sensitivity to memory latency and cache utilization. We study the cache utilization of a warp by determining its *hit ratio*, the percentage of memory requests that hit in the cache when the warp issues a single memory instruction. As Figure 2 shows, the warps from each of our three representative GPGPU applications are distributed across all possible ranges of *hit ratio*, exhibiting significant heterogeneity. To better characterize warp behavior, we break the warps down into the five types shown in Figure 3 based on their hit ratios: *all-hit*, *mostly-hit*, *balanced*, *mostly-miss*, and *all-miss*.

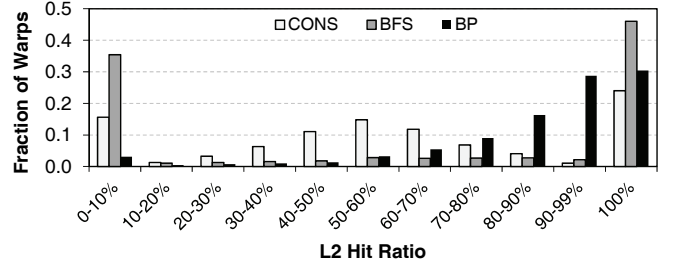


Figure 2: L2 cache hit ratio of different warps in three representative GPGPU applications. Reproduced from [6].

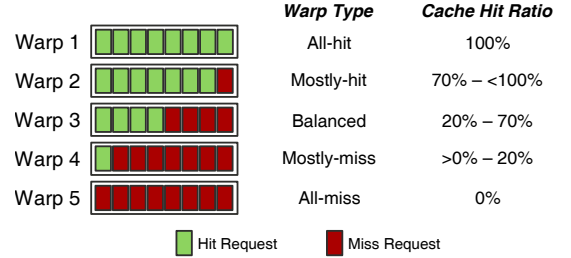


Figure 3: Warp type categorization based on the shared cache hit ratios. Hit ratio values are empirically chosen. Reproduced from [6].

MeDiC provide two mechanisms, warp-type-aware cache bypassing and warp-type-aware cache insertion policy, in order to convert *mostly-hit* warps into *all-hit* warps, where all requests in the warp hit in the cache, thereby reducing the stall time of *mostly-hit* warp significantly. This is done at the cost of converting the *mostly-miss* warps into *all-miss* warps, but doing so does not increase the stall time of such warps. To speed up uncacheable cache misses from *mostly-hit* warps, the warp-type-aware memory scheduling policy in MeDiC prioritizes memory requests from *mostly-hit* warps over memory requests from *mostly-miss* warps.

2.2. Memory Divergence Stability Over Time

A warp tends to retain its memory divergence behavior (e.g., whether or not it is *mostly-hit* or *mostly-miss*) for long periods of execution, and is thus predictable. This is due to the spatial and temporal locality of each thread within the warp. Figure 4 shows a sample of warps from a representative application (i.e., BFS [10]) that shows this predictability. This predictability enables us to perform history-based warp divergence characterization.

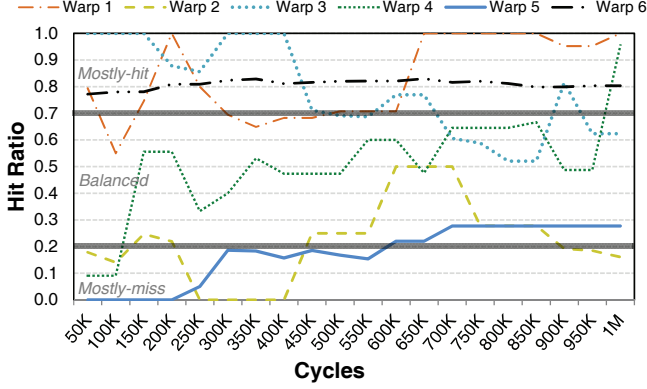


Figure 4: Hit ratio of randomly selected warps over time from BFS. Reproduced from [6].

2.3. High Queuing Latencies at the Shared Cache

Due to the amount of thread parallelism within a GPU, a large number of memory requests can arrive at the L2 cache in a small window of execution time, leading to significant queuing delays. Prior work observes high access latencies for the shared L2 cache within a GPU [126, 127, 142], but does not identify *why* these latencies are so high. We show that when a large number of requests arrive at the L2 cache, both the limited number of read/write ports and backpressure from cache bank conflicts force many of these requests to queue up for long periods of time. We observe that this queuing latency can sometimes add *hundreds* of cycles to the cache access latency, and that non-uniform queuing across the different cache banks exacerbates memory divergence. Figure 5 quantifies the magnitude of this queue contention if we set the cache lookup latency at one cycle, for one application, BFS [10]. As shown, a significant number of requests experience tens to hundreds of cycles of queuing delay.

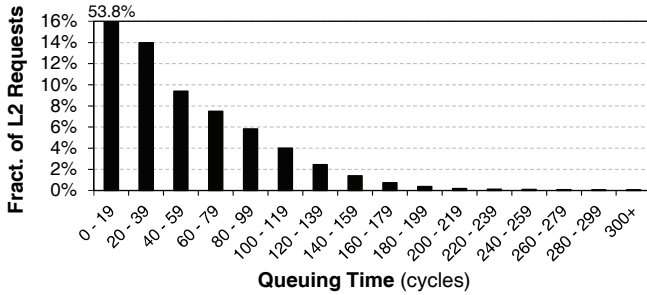


Figure 5: Distribution of per-request queuing latencies for L2 cache requests from BFS. Reproduced from [6].

The warp-type-aware bypassing logic in MeDiC helps to alleviate these L2 queuing latencies. By preventing mostly-miss and all-miss warps from accessing the cache, which yields little or no benefit to them, we reduce the access latencies for requests from (1) mostly-hit and all-hit warps, which benefit from the cache much more; and also (2) mostly-miss and all-miss warps themselves; thereby improving the overall performance of all warps and the system.

3. MeDiC: Memory Divergence Correction

Based on these three new observations we made, we define three major goals for our new mechanism. We would like to devise a mechanism that (1) converts mostly-hit warps into *all-hit warps* (warps where *all* requests hit in the cache, as shown in Figure 1c), (2) converts mostly-miss warps into *all-miss warps* (warps where *none* of the requests hit in the cache, as shown in Figure 1d) and (3) reduces L2 cache queuing delay for all warp types. As we can see in Figure 1a, the stall time due to memory divergence for the mostly-hit warp can be eliminated by converting only the single cache miss (Figure 1a, ②) into a cache hit.

To this end, we introduce *Memory Divergence Correction* (MeDiC), a GPU-specific mechanism that exploits *memory divergence heterogeneity* across warps at the shared cache and at main memory to improve the overall performance of GPGPU applications. MeDiC consists of three different components, which work together to achieve our three goals: (1) a warp-type-aware *cache bypassing mechanism*, which prevents requests from mostly-miss and all-miss warps from accessing the shared L2 cache; (2) a warp-type-aware *cache insertion policy*, which prioritizes requests from mostly-hit and all-hit warps, in order to increase the likelihood that they all become cache hits; and (3) a warp-type-aware *memory scheduling mechanism*, which prioritizes requests from mostly-hit warps that were not successfully converted to all-hit warps, in order to minimize the stall time due to divergence. These three components are all driven by an online mechanism that can identify the expected memory divergence behavior of each warp.

Figure 6 shows the overall MeDiC mechanism. MeDiC consists of four different components: ① a *warp-type-identification mechanism* that classifies warps into one of the four warp types as described in Section 2.1; ② a *bypass mechanism* that bypasses requests from all-miss and mostly-miss warps, reducing the queuing delay in the L2 cache; ③ an *insertion policy* that prevent mostly-hit requests from being evicted from the cache; and ④ a *memory scheduler* that prioritizes requests from mostly-hit warps, which are more latency sensitive.

3.1. Warp Type Identification

In order to take advantage of the memory divergence heterogeneity across warps, we must first add hardware that can identify the divergence behavior of each warp. The key idea is to periodically sample the hit ratio of a warp, and to classify the warp’s divergence behavior as one of the five types in Figure 3 based on the observed hit ratio. This information can then be used to drive the warp-type-aware components of MeDiC. In general, warps tend to retain the same memory divergence behavior for long periods of execution. However, there can be some long-term shifts in warp divergence behavior, requiring periodic resampling of the hit ratio to potentially re-evaluate the warp type. Warp type

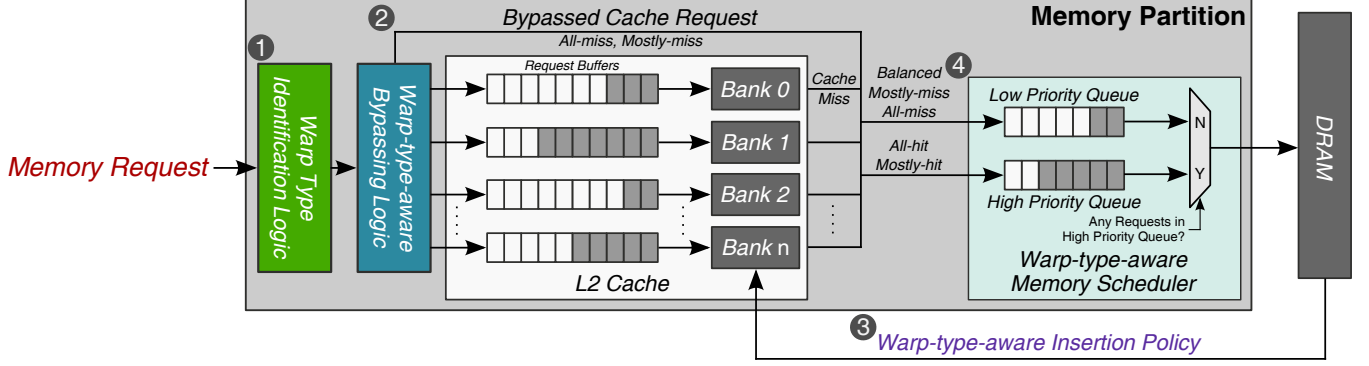


Figure 6: Overview of MeDiC: ① warp type identification logic, ② warp-type-aware cache bypassing, ③ warp-type-aware cache insertion policy, ④ warp-type-aware memory scheduler. Reproduced from [6].

identification through hit ratio sampling requires hardware within the cache to periodically count the number of hits and misses each warp incurs. We append two counters to the metadata stored for each warp, which represent the total number of cache hits and cache accesses for the warp during the sampling interval.

3.2. Warp-type-aware Shared Cache Bypassing

Once the warp type is known and a warp generates a request to the L2 cache, our mechanism first decides whether to bypass the cache based on the warp type. The key idea behind *warp-type-aware cache bypassing* is to convert mostly-miss warps into all-miss warps, as they do not benefit greatly from the few cache hits that they get. By bypassing these requests, we achieve three benefits: (1) bypassed requests can avoid L2 queuing latencies entirely, (2) other requests that do hit in the L2 cache experience shorter queuing delays due to the reduced contention, and (3) space is created in the L2 cache for mostly-hit warps.

The cache bypassing logic must make a simple decision: if an incoming memory request was generated by a mostly-miss or all-miss warp, the request is bypassed directly to DRAM. This is determined by reading the warp type stored in the warp metadata from the warp type identification mechanism. A simple 2-bit demultiplexer can be used to determine whether a request is sent to the L2 bank arbiter, or directly to the DRAM request queue.

3.3. Warp-type-aware Cache Insertion Policy

Our cache bypassing mechanism frees up space within the L2 cache, which we want to use for the cache misses from mostly-hit warps (to convert the cache miss memory requests into cache hits). However, even with the new bypassing mechanism, other warps (e.g., balanced, mostly-miss) still insert some data into the cache. In order to aid the conversion of mostly-hit warps into all-hit warps, we develop a *warp-type-aware cache insertion policy*, whose key idea is to ensure that in a given cache set, data from mostly-miss warps are

evicted first, while data from mostly-hit warps and all-hit warps are evicted last.

To ensure that a cache block from a mostly-hit warp stays in the cache for as long as possible, we insert the block closer to the MRU position. A cache block requested by a mostly-miss warp is inserted closer to the LRU position, making it more likely to be evicted. To track the warp type associated with these cache blocks, we add two bits of metadata to each cache block, indicating the warp type. These bits are then appended to the replacement policy bits. The bits modify the replacement policy behavior, such that a cache block from a mostly-miss warp is more likely to get evicted than a block from a balanced warp. Similarly, a cache block from a balanced warp is more likely to be evicted than a block from a mostly-hit or all-hit warp.

3.4. Warp-type-aware Memory Scheduler

Our cache bypassing mechanism and cache insertion policy work to increase the likelihood that *all* requests from a mostly-hit warp become cache hits, converting the warp into an all-hit warp. However, due to cache conflicts, or due to poor locality, there may still be cases when a mostly-hit warp cannot be fully converted into an all-hit warp, and is therefore unable to avoid stalling due to memory divergence as at least one of its requests has to go to DRAM. In such a case, we want to minimize the amount of time that this warp stalls. To this end, we propose a *warp-type-aware memory scheduler* that prioritizes the occasional DRAM requests from mostly-hit warps.

The design of our memory scheduler is very simple. Each memory request is tagged with a single bit, which is set if the memory request comes from a mostly-hit warp (or an all-hit warp, in case the warp was mischaracterized). We modify the request queue at the memory controller to contain two different queues, where a *high-priority queue* contains all requests that have their mostly-hit bit set to one. The *low-priority queue* contains all other requests, whose mostly-hit bits are set to zero. Each queue uses FR-FCFS [115, 156] as

the scheduling policy; however, the scheduler always selects requests from the high priority queue over requests in the low priority queue.¹

We describe each component of MeDiC in more detail in Sections 4.1, 4.2, 4.3 and 4.4 of our PACT 2015 paper [6].

4. Methodology

We model our mechanism using GPGPU-Sim 3.2.1 [9]. We modified GPGPU-Sim to accurately model cache bank conflicts, and added the cache bypassing, cache insertion, and memory scheduling mechanisms needed to support MeDiC. We use GPUWattch [76] to evaluate power consumption. We have open sourced our simulator source code at [118]. We evaluate our system across multiple GPGPU applications from the CUDA SDK [102], Rodinia [19], MARS [39], and Lonestar [10] benchmark suites.

We report performance results using the harmonic average of the IPC speedup (over the baseline GPU) of each kernel of each application.² Harmonic speedup [28, 85] was shown to reflect the average normalized execution time in multi-programmed workloads. We calculate energy efficiency for each workload by dividing the IPC by the energy consumed. Section 5 of our PACT 2015 paper [6] provides more detail on our experimental methodology.

5. Evaluation

Figure 7 shows the performance of MeDiC compared to four GPU cache management mechanisms: the Evicted Address Filter insertion policy [123] (**EAF**), PCAL bypassing policy [79] (**PCAL**), PC-based cache bypassing policy (**PC-ByP**) and an idealized random bypassing policy (**Rand**) over 15 different GPGPU applications from 4 benchmark suites. We also show the performance of each individual component of MeDiC: our warp-type-aware insertion policy (**WIP**), our warp-type-aware memory scheduling policy (**WMS**) and our warp-type-aware bypassing policy (**WByP**).

We found that each component of MeDiC individually provides significant performance improvement: WIP (32.5%), WMS (30.2%), and WByP (33.6%). MeDiC, which combines all three mechanisms, provides a 41.5% performance improvement over Baseline, on average. MeDiC matches or outperforms its individual components for all benchmarks except BP, where MeDiC has a higher L2 miss rate and lower row buffer locality than WMS and WByP.

Our insertion policy, WIP, outperforms EAF [123] by 12.2%. We observe that the key benefit of WIP is that cache blocks

from mostly-miss warps are much more likely to be evicted. In addition, WIP reduces the cache miss rate of several applications. Our memory scheduler, WMS, provides significant performance gains (30.2%) over Baseline, because the memory scheduler prioritizes requests from warps that have a high hit ratio, allowing these warps to become active much sooner than they do in Baseline. Our bypassing mechanism, WByP provides an average 33.6% performance improvement over Baseline, because it is effective at reducing the L2 queuing latency.

Compared to PCAL [79], WByP provides 12.8% better performance, and full MeDiC provides 21.8% better performance. We observe that while PCAL reduces the amount of cache thrashing, the reduction in thrashing does not directly translate into better performance. We observe that warps in the mostly-miss category sometimes have high reuse, and acquire tokens to access the cache. This causes less cache space to become available for mostly-hit warps, limiting how many of these warps become all-hit. However, when high-reuse warps that possess tokens are mainly in the mostly-hit category (PVC, PVR, SS, and BH), we find that PCAL performs better than WByP.

Compared to Rand,³ MeDiC performs 6.8% better, because MeDiC is able to make bypassing decisions that do not increase the miss rate significantly. This leads to lower off-chip bandwidth usage under MeDiC than under Rand. Rand increases the cache miss rate by 10% for the kernels of several applications (BP, PVC, PVR, BFS, and MST). We observe that in many cases, MeDiC improves the performance of applications that tend to generate a large number of memory requests, and thus experience substantial queuing latencies.

Compared to PC-ByP, MeDiC performs 12.4% better. We observe that the overhead of tracking the PC becomes significant, and that thrashing occurs as two PCs can hash to the same index, leading to inaccuracies in the bypassing decisions.

We conclude that each component of MeDiC, and the full MeDiC framework, are effective. Note that each component of MeDiC addresses the same problem (i.e., memory divergence of threads within a warp) using different techniques on different parts of the memory hierarchy. For the majority of workloads, one optimization is enough. However, we see that for certain high-intensity workloads (BFS and SSSP), the congestion is so high that we need to attack divergence on multiple fronts. Thus, MeDiC provides better average performance than all of its individual components, especially for such memory-intensive workloads.

We provide the following other evaluation results in our PACT 2015 paper [6]:

- Impact of MeDiC on cache miss rate.

¹Using two queues ensures that high-priority requests are not blocked by low-priority requests even when the low-priority queue is full. Two-queue priority also uses simpler logic design than comparator-based priority [5, 132, 133].

²We confirm that for each application, all kernels have similar speedup values, and that aside from SS and PVC, there are no outliers (i.e., no kernel has a much higher speedup than the other kernels). To verify that harmonic speedup is not swayed greatly by these few outliers, we recompute it for SS and PVC *without* these outliers, and find that the outlier-free speedup is within 1% of the harmonic speedup we report in the paper.

³Note that our evaluation uses an ideal random bypassing mechanism, where we manually select the best individual percentage of requests to bypass the cache for each workload. As a result, the performance shown for Rand is better than can be practically realized.

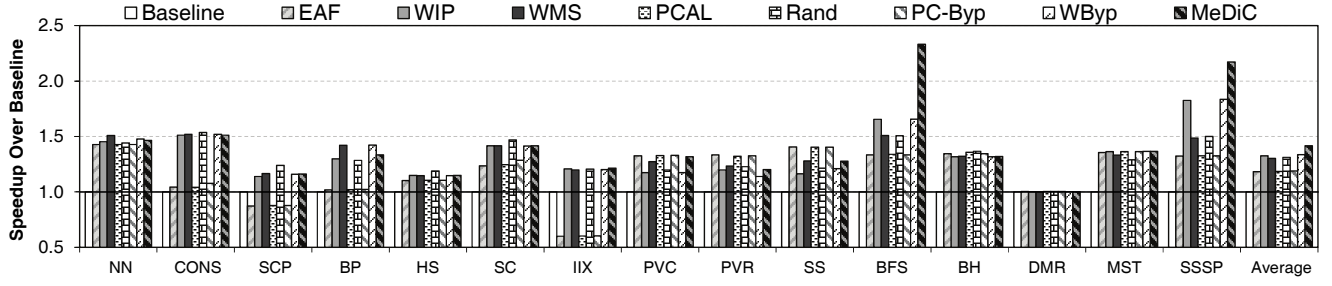


Figure 7: Performance of MeDiC. Adapted from [6].

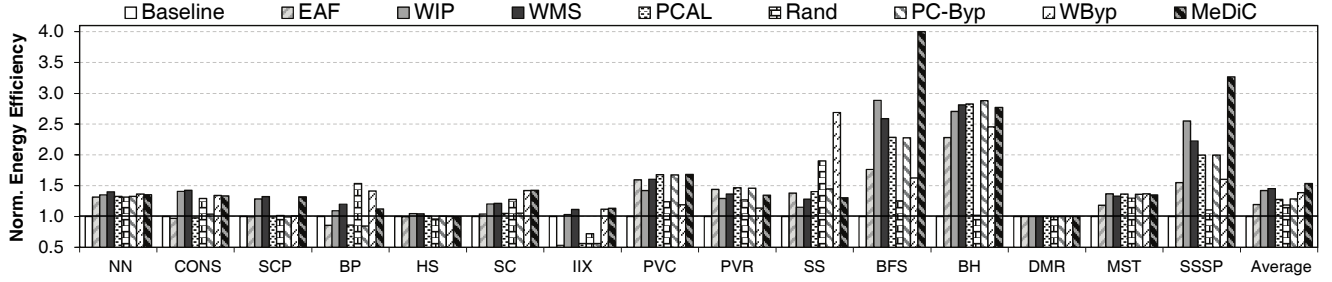


Figure 8: Energy efficiency of MeDiC. Adapted from [6].

- Impact of MeDiC on queuing latency.
- Impact of MeDiC on row buffer locality.
- Analysis of reuse in GPGPU applications.
- Hardware cost of MeDiC.

6. Related Work

To our knowledge, MeDiC is the first work that identifies inter-warp memory divergence heterogeneity and exploits it to achieve better system performance in GPGPU applications. Our new mechanism consists of warp-type-aware components for cache bypassing, cache insertion, and memory scheduling. We have already provided extensive quantitative and qualitative comparisons to state-of-the-art mechanisms in GPU cache bypassing [79], cache insertion [123], and memory scheduling [115, 156]. In this section, we discuss other related work in these areas.

Hardware-based Cache Bypassing. PCAL is a bypassing mechanism that addresses the cache thrashing problem by throttling the number of threads that time-share the cache at any given time [79]. The key idea of PCAL is to limit the number of threads that get to access the cache. Concurrent work by Li et al. [78] proposes a cache bypassing mechanism that allows only threads with high reuse to utilize the cache. The key idea is to use locality filtering based on the reuse characteristics of GPGPU applications, with only high reuse threads having access to the cache. Xie et al. [146] propose a bypassing mechanism at the thread block level. In their mechanism, the compiler statically marks whether thread blocks prefer caching or bypassing. At runtime, the mechanism dynamically selects a subset of thread blocks to use the cache, to increase cache utilization.

Chen et al. [20, 21] propose a combined warp throttling and bypassing mechanism for the L1 cache based on the cache-conscious warp scheduler [116]. The key idea is to bypass the cache when resource contention is detected. This is done by embedding history information into the L2 tag arrays. The L1 cache uses this information to perform bypassing decisions, and only warps with high reuse are allowed to access the L1 cache. Jia et al. propose an L1 bypassing mechanism [48], whose key idea is to bypass requests when there is an associativity stall. Dai et al. propose a mechanism to bypass cache based on a model of a cache miss rate [23].

MeDiC differs from these prior cache bypassing works because it uses warp memory divergence heterogeneity for bypassing decisions. We also show (in Section 6.4 of our PACT 2015 paper [6]) that our mechanism implicitly takes reuse information into account.

Software-based Cache Bypassing. Concurrent work by Li et al. [77] proposes a compiler-based technique that performs cache bypassing using a method similar to PCAL [79]. Xie et al. [145] propose a mechanism that allows the compiler to perform cache bypassing for global load instructions. Both of these mechanisms are different from MeDiC in that MeDiC applies bypassing to *all* loads and stores that utilize the shared cache, without requiring additional characterization at the compiler level. Mekkat et al. [87] propose a bypassing mechanism for when a CPU and a GPU share the last level cache. Their key idea is to bypass GPU cache accesses when CPU applications are cache sensitive, which is not applicable to GPU-only execution.

CPU Cache Bypassing. There are also several other CPU-based cache bypassing techniques. These techniques include using additional buffers track cache statistics to predict cache

blocks that have high utility based on reuse count [18, 27, 32, 50, 55, 81, 144, 152], reuse distance [18, 24, 29, 31, 34, 104, 143, 149], behavior of the cache block [46] or miss rate [22, 88, 120, 137]. As they do not operate on SIMD systems, these mechanisms do not (need to) account for memory divergence heterogeneity when performing bypassing decisions.

Cache Insertion and Replacement Policies. Many works propose different insertion policies for CPU systems (e.g., [44, 45, 54, 110, 112, 123]). We compare our insertion policy against the Evicted-Address Filter (EAF) [123], and show that our policy, which takes advantage of inter-warp divergence heterogeneity, outperforms EAF. Dynamic Insertion Policy (DIP) [44] and Dynamic Re-Reference Interval Prediction (DRRIP) [45] are insertion policies that account for cache thrashing. The downside of these two policies is that they are unable to distinguish between high-reuse and low-reuse blocks in the same thread [123]. The Bi-modal Insertion Policy [110] dynamically characterizes the cache blocks being inserted. None of these works take warp type characteristics or memory divergence behavior into account. Other work proposed prefetch-aware insertion and replacement policies [25, 124, 130]. MeDiC can be combined with such policies.

Memory Scheduling. Yuan et al. propose a GPU interconnect design that rearrange the sequence of memory requests that arrive at each memory channel to reduce the complexity of GPU memory scheduler [151]. Chatterjee et al. propose a GPU memory scheduler that allows requests from the same warp to be grouped together, in order to reduce the memory divergence across different memory requests within the same warp [17]. Jog et al. propose a GPU memory scheduler that exploit the criticality information of warps in the GPU cores in order to improve the performance of GPGPU applications [49]. Principles of MeDiC can be incorporated into these schedulers.

There are several memory scheduler designs that target systems with CPUs [26, 33, 43, 56, 57, 59, 60, 67, 68, 69, 82, 93, 94, 95, 96, 98, 99, 115, 131, 132, 133, 134, 135, 147, 153], and heterogeneous compute elements [5, 47, 138]. Memory schedulers for CPUs and heterogeneous systems generally aim to reduce interference across different applications.

Improving DRAM. An alternative approach to mitigate memory divergence is to improve the performance of the main memory itself. Previous works propose new DRAM designs that are capable of reducing memory latency in conventional DRAM [1, 2, 3, 4, 11, 12, 13, 13, 14, 14, 15, 16, 35, 36, 37, 38, 40, 41, 42, 53, 58, 61, 70, 71, 72, 73, 74, 83, 86, 92, 97, 100, 103, 109, 119, 121, 122, 125, 129, 141, 154] as well as non-volatile memory [62, 64, 65, 66, 80, 84, 90, 91, 111, 113, 114, 148, 150]. Data compression techniques can increase the effective DRAM bandwidth [105, 106, 107, 108, 140]. All these techniques are orthogonal to MeDiC and can be used to further improve the performance of GPGPU applications.

Other Ways to Handle Memory Divergence. In addition to cache bypassing, cache insertion policy, and memory scheduling, other works propose different methods of decreasing memory divergence [51, 52, 63, 75, 89, 101, 116, 117, 155]. These methods range from thread throttling [51, 52, 63, 116] to warp scheduling [75, 89, 101, 116, 117, 155]. While these methods share our goal of reducing memory divergence, none of them exploit inter-warp heterogeneity and, as a result, are either orthogonal or complementary to our proposal. Our work also makes new observations about memory divergence not covered by these works.

7. Potential Impact

While the problem that MeDiC is trying to solve, which is memory divergence, is not new, key findings in this work provide novelty and create potential research topics for the future. We discuss at least three such opportunities and future directions.

Taking Advantage of Memory Divergence Heterogeneity. MeDiC modifies the memory hierarchy to introduce awareness of the memory divergence heterogeneity between different types of warps. There are many other applications that can exploit warp type information. Other resources within the GPU (e.g., GPU cores, warp scheduler) can exploit the memory divergence heterogeneity across different warps to further improve the performance of GPGPU applications. For example, the warp type information can be used by the warp scheduler and thread block scheduler to ensure that they do not schedule warps of the same type to execute at the same time, to limit the amount of cache contention that occurs. Incorporating the warp type information with other techniques, such as assist warps to relieve execution bottlenecks [140], can enable GPUs to utilize resources based on the type of warps the GPU is executing. For example, mostly-hit warps favor a mechanism that provides low memory latency, while mostly-miss warps might favor a mechanism that provides higher off-chip bandwidth. Memory divergence heterogeneity can also be used to assist GPU resource virtualization [139], as virtual resource allocation can take into account the utilization of shared memory resources to determine how much of a particular memory resource to allocate to each thread block.

Warp type information can be used to improve the performance of GPU address translation. Prior works [7, 8] show that address translations that do not hit in a TLB can incur long-latency page table walks, which can affect hundreds of application threads at once. Such long-latency address translations might have a greater impact on warps that are latency sensitive (e.g., *mostly-hit* and *all-hit* warps). Thus, warp-type information can be combined with previously-proposed techniques that aim to reduce the overhead of GPU address translation [7, 8] to provide synergistic performance benefits.

We believe the idea of warp-type heterogeneity enables many different mechanisms to customize execution on a GPU

to achieve higher performance and energy efficiency. Hence, our PACT 2015 paper [6] paves the way for fine-grained customization of a GPU.

Identifying Long-Latency Threads in a Warp. Our PACT 2015 paper [6] shows how to intelligently reduce the memory latency of threads within a warp in order to reduce the memory divergence problem. However, MeDiC focuses on reducing the stall time of mostly-hit warps. Long-latency threads can still exist in the mostly-hit warps due to other problems such as load balancing at the memory partitions. Additional work on (1) how to identify latency-critical threads within a warp and (2) how to accelerate these specific threads can further improve the performance and energy efficiency of GPGPU applications.

Reducing High Queuing Delays and Memory Contention in the GPU Memory Hierarchy. As shown in our PACT 2015 paper [6], the queuing delay of throughput processors such as GPUs can become a performance bottleneck, as the delay increases the stall time of warps of *all* types. While the proposed warp-type-aware cache bypassing mechanism in MeDiC aims to reduce the queuing delay, non-uniform memory access patterns can still cause contention at a few L2 cache banks and memory partitions. In future systems, the parallelism of throughput processors is likely to increase further. For example, future GPUs will likely come with a higher number of GPU cores and larger SIMD widths. This is expected to greatly increase the amount of contention and, thus, queuing delay, for many resources. The different components of MeDiC can serve as a starting point for future research on alleviating cache and memory contention in future systems, and can ultimately enable a larger amount of thread-level parallelism. We believe studying the mitigation of high cache and memory contention is very promising for future parallel throughput processors and encourage future work in this area.

8. Conclusion

Warps from GPGPU applications exhibit heterogeneity in their memory divergence behavior at the shared L2 cache within the GPU. We find that (1) some warps benefit significantly from the cache, while others make poor use of it; (2) such divergence behavior for a warp tends to remain stable for long periods of the warp’s execution; and (3) the impact of memory divergence can be amplified by the high queuing latencies at the L2 cache.

We propose *Memory Divergence Correction* (MeDiC), whose key idea is to identify memory divergence heterogeneity online in hardware and use this information to drive cache management and memory scheduling, by prioritizing warps that take the greatest advantage of the shared cache. To achieve this, MeDiC consists of three *warp-type-aware* components for (1) cache bypassing, (2) cache insertion, and (3) memory scheduling. MeDiC delivers significant performance and energy improvements over multiple previously proposed

policies, and over a state-of-the-art GPU cache management technique. We conclude that exploiting inter-warp heterogeneity is effective, and hope future works explore other ways of improving systems based on this key observation of our work.

Acknowledgments

We thank the anonymous reviewers and SAFARI group members for their feedback. Special thanks to Mattan Erez for his valuable feedback on our PACT 2015 paper. We acknowledge the support of our industrial partners: Facebook, Google, IBM, Intel, Microsoft, NVIDIA, Qualcomm, VMware, and Samsung. This research was partially supported by the NSF (grants 0953246, 1065112, 1205618, 1212962, 1213052, 1302225, 1302557, 1317560, 1320478, 1320531, 1409095, 1409723, 1423172, 1439021, and 1439057), the Intel Science and Technology Center for Cloud Computing, and the Semiconductor Research Corporation.

References

- [1] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, “A Scalable Processing-in-memory Accelerator for Parallel Graph Processing,” in *ISCA*, 2015.
- [2] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, “PIM-enabled Instructions: A Low-overhead, Locality-aware Processing-in-memory Architecture,” in *ISCA*, 2015.
- [3] J. H. Ahn, J. Leverich, R. Schreiber, and N. P. Jouppi, “Multicore DIMM: an Energy Efficient Memory Module with Independently Controlled DRAMs,” *IEEE CAL*, 2009.
- [4] J. H. Ahn, N. P. Jouppi, C. Kozyrakis, J. Leverich, and R. S. Schreiber, “Improving System Energy Efficiency with Memory Rank Subsetting,” *ACM TACO*, vol. 9, no. 1, pp. 4:1–4:28, 2012.
- [5] R. Ausavarungnirun, K. K. Chang, L. Subramanian, G. Loh, and O. Mutlu, “Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems,” in *ISCA*, 2012.
- [6] R. Ausavarungnirun, S. Ghose, O. Kayiran, G. H. Loh, C. R. Das, M. T. Kandemir, and O. Mutlu, “Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance,” in *PACT*, 2015.
- [7] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu, “Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes,” in *MICRO*, 2017.
- [8] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. Rossbach, and O. Mutlu, “MASK: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency,” in *ASPLOS*, 2018.
- [9] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, “Analyzing CUDA Workloads Using a Detailed GPU Simulator,” in *ISPASS*, 2009.
- [10] M. Burtcher, R. Nasre, and K. Pingali, “A Quantitative Study of Irregular Programs on GPUs,” in *IISWC*, 2012.
- [11] K. Chandrasekar, S. Goossens, C. Weis, M. Koedam, B. Akesson, N. Wehn, and K. Goossens, “Exploiting Expendable Process-Margins in DRAMs for Run-Time Performance Optimization,” in *DATE*, 2014.
- [12] K. K. Chang, A. Kashyap, H. Hassan, S. Ghose, K. Hsieh, D. Lee, T. Li, G. Pekhimenko, S. Khan, and O. Mutlu, “Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization,” in *SIGMETRICS*, 2016.
- [13] K. K. Chang, D. Lee, Z. Chishti, A. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu, “Improving DRAM Performance by Parallelizing Refreshes with Accesses,” in *HPCA*, 2014.
- [14] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu, “Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM,” in *HPCA*, 2016.
- [15] K. K. Chang, A. G. Yaglikci, S. Ghose, A. Agrawal, N. Chatterjee, A. Kashyap, D. Lee, M. O’Connor, H. Hassan, and O. Mutlu, “Understanding Reduced-Voltage Operation in Modern DRAM Devices: Experimental Characterization, Analysis, and Mechanisms,” in *SIGMETRICS*, 2017.
- [16] N. Chatterjee, M. Shevgoor, R. Balasubramanian, A. Davis, Z. Fang, R. Illikkal, and R. Iyer, “Leveraging Heterogeneity in DRAM Main Memories to Accelerate Critical Word Access,” in *MICRO*, 2012.
- [17] N. Chatterjee, M. O’Connor, G. H. Loh, N. Jayasena, and R. Balasubramanian, “Managing DRAM Latency Divergence in Irregular GPGPU Applications,” in *SC*, 2014.
- [18] M. Chaudhuri, J. Gaur, N. Bashyam, S. Subramoney, and J. Nuzman, “Introducing Hierarchy-awareness in Replacement and Bypass Algorithms for Last-Level Caches,” in *PACT*, 2012.

- [19] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IISWC*, 2009.
- [20] X. Chen, L.-W. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W. W. Hwu, "Adaptive Cache Management for Energy-Efficient GPU Computing," in *MICRO*, 2014.
- [21] X. Chen, S. Wu, L.-W. Chang, W.-S. Huang, C. Pearson, Z. Wang, and W. W. Hwu, "Adaptive Cache Bypass and Insertion for Many-Core Accelerators," in *MES*, 2014.
- [22] J. D. Collins and D. M. Tullsen, "Hardware Identification of Cache Conflict Misses," in *MICRO*, 1999.
- [23] H. Dai, C. Li, H. Zhou, S. Gupta, C. Kartsaklis, and M. Mantor, "A Model-driven Approach to Warp/thread-block Level GPU Cache Bypassing," in *DAC*, 2016.
- [24] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum, "Improving Cache Management Policies Using Dynamic Reuse Distances," in *MICRO*, 2012.
- [25] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Prefetch-aware Shared Resource Management for Multi-core Systems," in *ISCA*, 2011.
- [26] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt, "Parallel Application Memory Scheduling," in *MICRO*, 2011.
- [27] Y. Etsion and D. G. Feitelson, "Exploiting Core Working Sets to Filter the L1 Cache with Random Sampling," *IEEE TC*, vol. 61, no. 11, pp. 1535–1550, 2012.
- [28] S. Eyerhan and L. Eeckhout, "System-Level Performance Metrics for Multiprogram Workloads," *IEEE Micro*, 2008.
- [29] M. Feng, C. Tian, and R. Gupta, "Enhancing LRU Replacement via Phantom Associativity," in *INTERACT*, Feb 2012.
- [30] M. Flynn, "Very High-Speed Computing Systems," *Proc. of the IEEE*, vol. 54, no. 2, 1966.
- [31] H. Gao and C. Wilkerson, "A Dueling Segmented LRU Replacement Algorithm with Adaptive Bypassing," in *JWAC*, 2010.
- [32] J. Gaur, M. Chaudhuri, and S. Subramoney, "Bypass and Insertion Algorithms for Exclusive Last-Level Caches," in *ISCA*, 2011.
- [33] S. Ghose, H. Lee, and J. F. Martinez, "Improving Memory Scheduling via Processor-side Load Criticality Information," in *ISCA*, 2013.
- [34] S. Gupta, H. Gao, and H. Zhou, "Adaptive Cache Bypassing for Inclusive Last Level Caches," in *IPDPS*, 2013.
- [35] C. A. Hart, "CDRAM in a Unified Memory Architecture," in *Intl. Computer Conference*, 1994.
- [36] M. Hashemi, O. Mutlu, and Y. N. Patt, "Continuous Runahead: Transparent Hardware Acceleration for Memory Intensive Workloads," in *MICRO*, 2016.
- [37] H. Hassan, G. Pekhimenko, N. Vijaykumar, V. Seshadri, D. Lee, O. Ergin, and O. Mutlu, "ChargeCache: Reducing DRAM Latency by Exploiting Row Access Locality," in *HPCA*, 2016.
- [38] H. Hassan, N. Vijaykumar, S. Khan, S. Ghose, K. K. Chang, G. Pekhimenko, D. Lee, O. Ergin, and O. Mutlu, "SoftMC: A Flexible and Practical Open-source Infrastructure for Enabling Experimental DRAM Studies," in *HPCA*, 2017.
- [39] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: A MapReduce Framework on Graphics Processors," in *PACT*, 2008.
- [40] H. Hidaka, Y. Matsuda, M. Asakura, and K. Fujishima, "The Cache DRAM Architecture," *IEEE Micro*, 1990.
- [41] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu, "Accelerating Pointer Chasing in 3D-stacked Memory: Challenges, Mechanisms, Evaluation," in *ICCD*, 2016.
- [42] W.-C. Hsu and J. E. Smith, "Performance of Cached DRAM Organizations in Vector Supercomputers," in *ISCA*, 1993.
- [43] E. Ipek, O. Mutlu, J. F. Martinez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," in *ISCA*, 2008.
- [44] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer, "Adaptive Insertion Policies for Managing Shared Caches," in *PACT*, 2008.
- [45] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP)," in *ISCA*, 2010.
- [46] J. Jalminger and P. Stenstrom, "A Novel Approach to Cache Block Reuse Predictions," in *ICPP*, 2003.
- [47] M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver, "A QoS-Aware Memory Controller for Dynamically Balancing GPU and CPU Bandwidth Use in an MPSoC," in *DAC*, 2012.
- [48] W. Jia, K. A. Shaw, and M. Martonosi, "MRPB: Memory Request Prioritization for Massively Parallel Processors," in *HPCA*, 2014.
- [49] A. Jog, O. Kayiran, A. Pattnaik, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Exploiting Core Criticality for Enhanced GPU Performance," in *SIGMETRICS*, 2016.
- [50] L. K. John and A. Subramanian, "Design and Performance Evaluation of A Cache Assist to Implement Selective Caching," in *ICCD*, 1997.
- [51] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither More Nor Less: Optimizing Thread-Level Parallelism for GPGPUs," in *PACT*, 2013.
- [52] O. Kayiran, N. C. Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das, "Managing GPU Concurrency in Heterogeneous Architectures," in *MICRO*, 2014.
- [53] G. Kedem and R. P. Koganti, "WCDRAM: A Fully Associative Integrated Cached-DRAM with Wide Cache Lines," *CS-1997-03, Duke*, 1997.
- [54] S. Khan, A. R. Alameldeen, C. Wilkerson, O. Mutlu, and D. A. Jimenez, "Improving Cache Performance using Read-write Partitioning," in *HPCA*, 2014.
- [55] M. Kharbutli and Y. Solihin, "Counter-Based Cache Replacement and Bypassing Algorithms," *IEEE TC*, vol. 57, no. 4, pp. 433–447, Apr. 2008.
- [56] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, "Bounding Memory Interference Delay in COTS-based Multi-core Systems," in *RTAS*, 2014.
- [57] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, "Bounding and Reducing Memory Interference in COTS-based Multi-core Systems," *Real-Time Systems*, vol. 52, no. 3, May 2016.
- [58] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," *CAL*, 2015.
- [59] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers," in *HPCA*, 2010.
- [60] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *MICRO*, 2010.
- [61] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM," in *ISCA*, 2012.
- [62] E. Kultursay, M. Kandemir, A. Sivasubramanian, and O. Mutlu, "Evaluating STT-RAM as an energy-efficient main memory alternative," in *ISPASS*, 2013.
- [63] H.-K. Kuo, B. C. Lai, and J.-Y. Jou, "Reducing Contention in Shared Last-Level Cache for Throughput Processors," *ACM TODAES*, vol. 20, no. 1, 2014.
- [64] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting Phase Change Memory as a Scalable DRAM Alternative," in *ISCA*, 2009.
- [65] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Phase Change Memory Architecture and the Quest for Scalability," *CACM*, vol. 53, no. 7, pp. 99–106, 2010.
- [66] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-Change Technology and the Future of Main Memory," *IEEE Micro*, vol. 30, no. 1, pp. 143–143, 2010.
- [67] C. J. Lee, V. Narasiman, O. Mutlu, and Y. N. Patt, "Improving Memory Bank-Level Parallelism in the Presence of Prefetching," in *MICRO*, 2009.
- [68] C. J. Lee, E. Ebrahimi, V. Narasiman, O. Mutlu, and Y. N. Patt, "DRAM-Aware Last-Level Cache Writeback: Reducing Write-Caused Interference in Memory Systems," Univ. of Texas at Austin, High Performance Systems Group, Tech. Rep. TR-HPS-2010-002, 2010.
- [69] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt, "Prefetch-aware DRAM Controllers," in *MICRO*, 2008.
- [70] D. Lee, S. Ghose, G. Pekhimenko, S. Khan, and O. Mutlu, "Simultaneous Multi-layer Access: Improving 3D-stacked Memory Bandwidth at Low Cost," *ACM TACO*, vol. 12, no. 4, p. 63, 2016.
- [71] D. Lee, S. Khan, L. Subramanian, S. Ghose, R. Ausavarungnirun, G. Pekhimenko, V. Seshadri, and O. Mutlu, "Design-Induced Latency Variation in Modern DRAM Chips: Characterization, Analysis, and Latency Reduction Mechanisms," in *SIGMETRICS*, 2017.
- [72] D. Lee, Y. Kim, G. Pekhimenko, S. Khan, V. Seshadri, K. K. Chang, and O. Mutlu, "Adaptive-latency DRAM: Optimizing DRAM Timing for the Common-case," in *HPCA*, 2015.
- [73] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu, "Tiered-latency DRAM: A Low Latency and Low Cost DRAM Architecture," in *HPCA*, 2013.
- [74] D. Lee, L. Subramanian, R. Ausavarungnirun, J. Choi, and O. Mutlu, "Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM," in *PACT*, 2015.
- [75] S.-Y. Lee and C.-J. Wu, "CAWS: Criticality-Aware Warp Scheduling for GPGPU Workloads," in *PACT*, 2014.
- [76] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: Enabling Energy Optimizations in GPGPUs," in *ISCA*, 2013.
- [77] A. Li, G.-J. van den Braak, A. Kumar, and H. Corporaal, "Adaptive and Transparent Cache Bypassing for GPUs," in *SC*, 2015.
- [78] C. Li, S. L. Song, H. Dai, A. Sidelnik, S. K. S. Hari, and H. Zhou, "Locality-Driven Dynamic GPU Cache Bypassing," in *ICS*, 2015.
- [79] D. Li, M. Rhu, D. Johnson, M. O'Connor, M. Erez, D. Burger, D. Fussell, and S. Redder, "Priority-Based Cache Allocation in Throughput Processors," in *HPCA*, 2015.
- [80] Y. Li, S. Ghose, J. Choi, J. Sun, H. Wang, and O. Mutlu, "Utility-Based Hybrid Memory Management," in *CLUSTER*, 2017.
- [81] H. Liu, M. Ferdman, J. Huh, and D. Burger, "Cache Bursts: A New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency," in *MICRO*, 2008.
- [82] W. Liu, P. Huang, T. Kun, T. Lu, K. Zhou, C. Li, and X. He, "LAMs: A Latency-aware Memory Scheduling Policy for Modern DRAM Systems," in *IPCCC*, 2016.
- [83] G. H. Loh, "3D-stacked Memory Architectures for Multi-core Processors," in *ISCA*, 2008.
- [84] Y. Lu, J. Shu, L. Sun, and O. Mutlu, "Loose-Ordering Consistency for Persistent Memory," in *ICCD*, 2014.
- [85] K. Luo, J. Gummaraju, and M. Franklin, "Balancing Throughput and Fairness in SMT Processors," in *ISPASS*, 2001.
- [86] Y. Luo, S. Govindan, B. Sharma, M. Santaniello, J. Meza, A. Kansal, J. Liu, B. Khesib, K. Vaid, and O. Mutlu, "Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-Reliability Memory," in *DSN*, 2014.
- [87] V. Mekkat, A. Holey, P.-C. Yew, and A. Zhai, "Managing Shared Last-Level Cache in a Heterogeneous Multicore Processor," in *PACT*, 2013.

- [88] G. Memik, G. Reinman, and W. H. Mangione-Smith, "Just Say No: Benefits of Early Cache Miss Determination," in *HPCA*, 2003.
- [89] J. Meng, D. Tarjan, and K. Skadron, "Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance," in *ISCA*, 2010.
- [90] J. Meza, Y. Luo, S. Khan, J. Zhao, Y. Xie, and O. Mutlu, "A Case for Efficient Hardware/Software Cooperative Management of Storage and Memory," in *WEED*, 2013.
- [91] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan, "Enabling Efficient and Scalable Hybrid Memories Using Fine-Granularity DRAM Cache Management," *IEEE CAL*, 2012.
- [92] Micron Technology, Inc., "576Mb: x18, x36 RLD RAM3," 2011.
- [93] T. Moscibroda and O. Mutlu, "Memory Performance Attacks: Denial of Memory Service in Multi-core Systems," in *USENIX Security*, 2007.
- [94] T. Moscibroda and O. Mutlu, "Distributed Order Scheduling and Its Application to Multi-core DRAM Controllers," in *PODC*, 2008.
- [95] J. Mukundan and J. F. Martinez, "MORSE: Multi-objective Reconfigurable Self-optimizing Memory Scheduler," in *HPCA*, 2012.
- [96] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, "Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning," in *MICRO*, 2011.
- [97] O. Mutlu, "Memory Scaling: A Systems Architecture Perspective," in *IMW*, 2013.
- [98] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *MICRO*, 2007.
- [99] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems," in *ISCA*, 2008.
- [100] O. Mutlu and L. Subramanian, "Research Problems and Opportunities in Memory Systems," *SUPERFRI*, 2014.
- [101] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling," in *MICRO*, 2011.
- [102] NVIDIA Corp., "CUDA C/C++ SDK Code Samples," <http://developer.nvidia.com/cuda-cc-sdk-code-samples>, 2011.
- [103] S. O. Y. H. Son, N. S. Kim, and J. H. Ahn, "Row-Buffer Decoupling: A Case for Low-Latency DRAM Microarchitecture," in *ISCA*, 2014.
- [104] J. Park, R. M. Yoo, D. S. Khudia, C. J. Hughes, and D. Kim, "Location-aware Cache Management for Many-core Processors with Deep Cache Hierarchy," in *SC*, 2013.
- [105] G. Pekhimenko, E. Bolotin, N. Vijaykumar, O. Mutlu, T. C. Mowry, and S. W. Keckler, "A Case for Toggle-aware Compression for GPU Systems," in *HPCA*, 2016.
- [106] G. Pekhimenko, T. Huberty, R. Cai, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Exploiting Compressed Block Size as an Indicator of Future Reuse," in *HPCA*, 2015.
- [107] G. Pekhimenko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry, "Linearly Compressed Pages: A Main Memory Compression Framework with Low Complexity and Low Latency," in *MICRO*, 2013.
- [108] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate Compression: Practical Data Compression for On-chip Caches," in *PACT*, 2012.
- [109] S. Phadke and S. Narayanasamy, "MLP Aware Heterogeneous Memory System," in *DATE*, 2011.
- [110] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive Insertion Policies for High Performance Caching," in *ISCA*, 2007.
- [111] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing Lifetime and Security of PCM-based Main Memory with Start-gap Wear Leveling," in *MICRO*, 2009.
- [112] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A Case for MLP-Aware Cache Replacement," in *ISCA*, 2006.
- [113] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable High Performance Main Memory System Using Phase-change Memory Technology," in *ISCA*, 2009.
- [114] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu, "ThyNVM: Enabling software-transparent crash consistency in persistent memory systems," in *MICRO*, 2015.
- [115] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory Access Scheduling," in *ISCA*, 2000.
- [116] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *MICRO*, 2012.
- [117] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Divergence-Aware Warp Scheduling," in *MICRO*, 2013.
- [118] SAFARI Research Group, "MeDiC - GitHub Repository," <https://github.com/CMU-SAFARI/MeDiC>.
- [119] Y. Sato *et al.*, "Fast cycle RAM (FCRAM): A 20-ns Random Row Access, Pipelined Operating DRAM," in *VLSIC*, 1998.
- [120] V. Seshadri, A. Bhowmick, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "The Dirty-Block Index," in *ISCA*, 2014.
- [121] V. Seshadri *et al.*, "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization," in *ISCA*, 2013.
- [122] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-memory Accelerator for Bulk Bitwise Operations using Commodity DRAM Technology," in *MICRO*, 2017.
- [123] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry, "The Evicted-Address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing," in *PACT*, 2012.
- [124] V. Seshadri, S. Yedkar, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Mitigating Prefetcher-Caused Pollution Using Informed Caching Policies for Prefetched Blocks," *ACM TACO*, vol. 11, no. 4, pp. 51:1–51:22, 2015.
- [125] W. Shin, J. Yang, J. Choi, and L.-S. Kim, "NUAT: A Non-Uniform Access Time Memory Controller," in *HPCA*, 2014.
- [126] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt, "Cache Coherence for GPU Architectures," in *HPCA*, 2013.
- [127] SiSoftware, "Benchmarks : Measuring GP (GPU/APU) Cache and Memory Latencies," <http://www.sisoftware.net>, 2014.
- [128] B. J. Smith, "A Pipelined, Shared Resource MIMD Computer," in *ICPP*, 1978.
- [129] Y. H. Son, S. O. Y. Ro, J. W. Lee, and J. H. Ahn, "Reducing Memory Access Latency with Asymmetric DRAM Bank Organizations," in *ISCA*, 2013.
- [130] S. Srinath, O. Mutlu, H. Kim, and Y. Patt, "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers," in *HPCA*, 2007.
- [131] 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*, 2010.
- [132] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, "BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling," in *IEEE TPDS*, 2016.
- [133] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, "The Blacklisting Memory Scheduler: Achieving high performance and fairness at low cost," in *ICCD*, 2014.
- [134] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu, "The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-application Interference at Shared Caches and Main Memory," in *MICRO*, 2015.
- [135] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu, "MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems," in *HPCA*, 2013.
- [136] J. E. Thornton, "Parallel Operation in the Control Data 6600," in *AFIPS FJCC*, 1964.
- [137] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun, "A Modified Approach to Data Cache Management," in *MICRO*, 1995.
- [138] H. Usui *et al.*, "DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators," *ACM TACO*, 2016.
- [139] N. Vijaykumar, K. Hsieh, G. Pekhimenko, S. Khan, A. Shrestha, S. Ghose, A. Jog, P. B. Gibbons, and O. Mutlu, "Zorua: A Holistic Approach to Resource Virtualization in GPUs," in *MICRO*, 2016.
- [140] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungrun, C. Das, M. Kandemir, T. C. Mowry, and O. Mutlu, "A Case for Core-Assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps," in *ISCA*, 2015.
- [141] F. A. Ware and C. Hampel, "Improving Power and Data Efficiency with Threaded Memory Modules," in *ICCD*, 2006.
- [142] H. Wong, M.-M. Papadopoulos, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU Microarchitecture Through Microbenchmarking," in *ISPASS*, 2010.
- [143] Y. Wu, R. Rakvic, L.-L. Chen, C.-C. Miao, G. Chrysos, and J. Fang, "Compiler Managed Micro-cache Bypassing for High Performance EPIC Processors," in *MICRO*, 2002.
- [144] L. Xiang, T. Chen, Q. Shi, and W. Hu, "Less Reused Filter: Improving L2 Cache Performance via Filtering Less Reused Lines," in *ICS*, 2009.
- [145] X. Xie, Y. Liang, G. Sun, and D. Chen, "An Efficient Compiler Framework for Cache Bypassing on GPUs," in *ICCAD*, 2013.
- [146] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang, "Coordinated Static and Dynamic Cache Bypassing for GPUs," in *HPCA*, 2015.
- [147] D. Xiong, K. Huang, X. Jiang, and X. Yan, "Memory Access Scheduling Based on Dynamic Multilevel Priority in Shared DRAM Systems," *ACM TACO*, vol. 13, no. 4, Dec. 2016.
- [148] H. Yoon, J. Meza, R. Ausavarungrun, R. Harding, and O. Mutlu, "Row Buffer Locality Aware Caching Policies for Hybrid Memories," in *ICCD*, 2012.
- [149] B. Yu, J. Ma, T. Chen, and M. Wu, "Global Priority Table for Last-Level Caches," in *DASC*, 2011.
- [150] X. Yu, C. J. Hughes, N. Satish, O. Mutlu, and S. Devadas, "Banshee: Bandwidth-efficient DRAM Caching via Software/Hardware Cooperation," in *MICRO*, 2017.
- [151] G. Yuan, A. Bakhoda, and T. Aamodt, "Complexity Effective Memory Access Scheduling for Many-Core Accelerator Architectures," in *MICRO*, 2009.
- [152] C. Zhang, G. Sun, P. Li, T. Wang, D. Niu, and Y. Chen, "SBAC: A Statistics Based Cache Bypassing Method for Asymmetric-access Caches," in *ISPLED*, 2014.
- [153] J. Zhao, O. Mutlu, and Y. Xie, "FIRM: Fair and High-Performance Memory Control for Persistent Memory Systems," in *MICRO*, 2014.
- [154] H. Zheng, J. Lin, Z. Zhang, E. Gorbato, H. David, and Z. Zhu, "Mini-rank: Adaptive DRAM Architecture for Improving Memory Power Efficiency," in *MICRO*, 2008.
- [155] Z. Zheng, Z. Wang, and M. Lipasti, "Adaptive Cache and Concurrency Allocation on GPGPUs," *IEEE CAL*, 2014.
- [156] W. K. Zuravleff and T. Robinson, "Controller for a Synchronous DRAM That Maximizes Throughput by Allowing Memory Requests and Commands to Be Issued Out of Order," US Patent No. 5,630,096, 1997.

Mosaic: An Application-Transparent Hardware–Software Cooperative Memory Manager for GPUs

Rachata Ausavarungnirun¹ Joshua Landgraf² Vance Miller² Saugata Ghose¹
Jayneel Gandhi³ Christopher J. Rossbach^{2,3} Onur Mutlu^{4,1}

¹Carnegie Mellon University ²The University of Texas at Austin

³VMware Research ⁴ETH Zürich

This paper summarizes the idea and key contributions of Mosaic, which was published at MICRO 2017 [8], and examines the work’s significance and future potential. Contemporary discrete GPUs support rich memory management features such as virtual memory and demand paging. These features simplify GPU programming by providing a virtual address space abstraction similar to CPUs and eliminating manual memory management, but they introduce high performance overheads during (1) address translation and (2) page faults. A GPU relies on high degrees of thread-level parallelism (TLP) to hide memory latency. Address translation can undermine TLP, as a single miss in the translation lookaside buffer (TLB) invokes an expensive serialized page table walk that often stalls multiple threads. Demand paging can also undermine TLP, as multiple threads often stall while they wait for an expensive data transfer over the system I/O (e.g., PCIe) bus when the GPU demands a page.

In modern GPUs, we face a trade-off on how the page size used for memory management affects address translation and demand paging. The address translation overhead is lower when we employ a larger page size (e.g., 2MB large pages, compared with conventional 4KB base pages), which increases TLB coverage and thus reduces TLB misses. Conversely, the demand paging overhead is lower when we employ a smaller page size, which decreases the system I/O bus transfer latency. Support for multiple page sizes can help relax the page size trade-off so that address translation and demand paging optimizations work together synergistically. However, existing page coalescing (i.e., merging base pages into a large page) and splintering (i.e., splitting a large page into base pages) policies require costly base page migrations that undermine the benefits multiple page sizes provide. In this paper, we observe that GPGPU applications present an opportunity to support multiple page sizes without costly data migration, as the applications perform most of their memory allocation en masse (i.e., they allocate a large number of base pages at once). We show that this en masse allocation allows us to create intelligent memory allocation policies which ensure that base pages that are contiguous in virtual memory are allocated to contiguous physical memory pages. As a result, coalescing and splintering operations no longer need to migrate base pages.

We introduce Mosaic, a GPU memory manager that provides application-transparent support for multiple page sizes. Mosaic

uses base pages to transfer data over the system I/O bus, and allocates physical memory in a way that (1) preserves base page contiguity and (2) ensures that a large page frame contains pages from only a single memory protection domain. We take advantage of this allocation strategy to design a novel in-place page size selection mechanism that avoids data migration. This mechanism allows the TLB to use large pages, reducing address translation overhead. During data transfer, this mechanism enables the GPU to transfer only the base pages that are needed by the application over the system I/O bus, keeping demand paging overhead low. Our evaluations show that Mosaic reduces address translation overheads while efficiently achieving the benefits of demand paging, compared to a contemporary GPU that uses only a 4KB page size. Relative to a state-of-the-art GPU memory manager, Mosaic improves the performance of homogeneous and heterogeneous multi-application workloads by 55.5% and 29.7% on average, respectively, coming within 6.8% and 15.4% of the performance of an ideal TLB where all TLB requests are hits.

1. Introduction

Graphics Processing Units (GPUs) are used for an ever-growing range of application domains due to their capability to provide high throughput. GPUs provide a high amount of throughput but they require a different programming model than CPUs, making their general adoption difficult. Recent support within GPUs for *memory virtualization* features, such as a unified virtual address space [57,70], demand paging [73], and preemption [2,73], can ease programming by allowing developers to exploit key benefits such as application portability and multi-application execution.

Hardware-supported memory virtualization relies on address translation to map each virtual memory address to a physical address within the GPU memory. Address translation uses page-granularity virtual-to-physical mappings that are stored within a multi-level *page table*. To look up a mapping within the page table, the GPU performs a page table walk, where a page table walker traverses through each level of the page table in main memory until the walker locates the *page table entry* for the requested mapping in the last level of the table. GPUs with virtual memory support have *translation lookaside buffers* (TLBs), which cache page table entries and avoid the need to perform a page table walk for

the cached entries, thereby reducing the address translation latency.

State-of-the-art GPU memory virtualization provides support for *demand paging* [3, 57, 73, 81, 102]. In demand paging, all of the memory used by a GPU application does *not* need to be transferred to the GPU memory at the beginning of application execution. Instead, during application execution, when a GPU thread issues a memory request to a page that has not yet been allocated in the GPU memory, the GPU issues a *page fault*, at which point the data for that page is transferred over the off-chip system I/O bus (e.g., the PCIe bus [76] in contemporary systems) from the CPU memory to the GPU memory. The transfer requires a long latency due to its use of an off-chip bus. Once the transfer completes, the GPU runtime allocates a physical GPU memory address to the page, and the thread can complete its memory request.

GPU Virtualization Challenges. Two fundamental challenges prevent further adoption of virtualization in GPUs: (1) the address translation challenge, and (2) the demand paging challenge. The address translation challenge stems from a long latency process that consists of a series of *serialized* memory accesses required to traverse the page table [80, 81]. As many threads can access different data present in a single page, these serialized page walk accesses significantly limit GPU concurrency, by lowering *thread-level parallelism* (TLP) and thereby reducing the latency hiding capability of a GPU. *Translation lookaside buffers* (TLBs) can reduce the latency of address translation by caching recently-used address translation information. Unfortunately, as application working sets and DRAM capacity have increased in recent years, state-of-the-art TLB designs [80, 81] suffer from *poor TLB reach*, i.e., the TLB covers only a small fraction of the physical memory working set of an application. We found that the poor TLB reach has a detrimental effect on GPU performance, because a *single* TLB miss can stall *hundreds* of threads at once, undermining TLP within a GPU and significantly reducing performance [8, 61, 95].

Figure 1 shows the performance of two GPU-MMU designs: (1) a design that uses the base 4KB page size, and (2) a design that uses a 2MB large page size, where both designs have *no demand paging overhead* (i.e., the system I/O bus transfer takes zero cycles to transfer a page). We normalize the performance of the two designs to a GPU with an ideal TLB, where *all* TLB requests hit in the L1 TLB. Our full experimental methodology is described in detail in our MICRO 2017 paper [8]. We make two major observations from the figure.

First, compared to the ideal TLB, the GPU-MMU with 4KB base pages experiences an average performance loss of 48.1%. We observe that with 4KB base pages, a single TLB miss often stalls *many* of the warps, which undermines the latency hiding behavior of the SIMT execution model used by GPUs. Second, the figure shows that using a 2MB page size with the same number of TLB entries as the 4KB design allows applications to come within 2% of the ideal TLB performance.

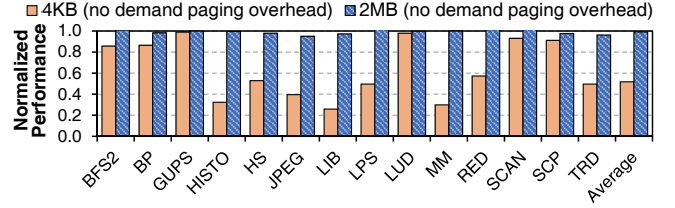


Figure 1: Performance of a GPU with *no demand paging overhead*, using (1) 4KB base pages and (2) 2MB large pages, normalized to the performance of a GPU with an ideal TLB. Reproduced from [8].

We find that with 2MB pages, the TLB has a much larger reach, which reduces the TLB miss rate substantially. Thus, there is strong incentive to use large pages for address translation.

To increase the TLB reach, *large pages* (e.g., the 2MB or 1GB pages used in many modern CPU architectures [39, 40]) can be employed. However, large pages increase the risk of *internal fragmentation*, where a portion of the large page is unallocated (or unused). Internal fragmentation occurs because it might often be difficult for an application to *completely* utilize large contiguous regions of memory. This fragmentation leads to (1) *memory bloat*, where a much greater amount of physical memory is allocated than the amount of memory that the application needs; and (2) longer memory access latencies, due to a potentially lower effective TLB reach and more page faults [56].

The demand paging challenge stems from a *page fault*, which requires a long-latency data transfer for an entire page over the system I/O bus [76]. Since GPU threads often access data in the same page due to data locality, a single page fault can cause *multiple* threads to stall at once. As a result, the page fault can significantly reduce the amount of TLP that the GPU can exploit, and thus significantly degrade performance [8, 102].

Unlike address translation, which benefits from *larger* pages, the overhead of demand paging is smaller when a *smaller* page size is used. A larger amount of data transfer increases the transfer time, increases the amount of time that GPU threads stall, and decreases TLP. Furthermore, as the size of a page increases, there is a greater probability that an application does *not* need all of the data in the page. As a result, threads may stall for a longer time without gaining any further benefit in return. Based on these two conflicting observations, memory virtualization in GPU systems has a fundamental trade-off due to the page size choice. We provide more detail on the trade-off in our MICRO 2017 paper [8].

2. Mosaic

In our MICRO 2017 paper [8], we propose *Mosaic*, a new GPU memory management scheme that aims to get the best of both small and large page sizes. *Mosaic* relaxes the page size trade-off by using *multiple* page sizes *transparently* to the application, and, thus, to the programmer. With multiple page sizes, and the ability to change virtual-to-physical mappings

dynamically, the GPU system can support *good TLB reach* by using large pages for address translation, while providing *good demand paging performance* by using base pages for data transfer. However, while coalescing multiple small pages into a large page requires a contiguous region, existing memory allocation mechanisms make it difficult to find regions of physical memory where base pages can be coalesced without a large number of page migration operations. This is because existing GPU memory allocation mechanisms do not allocate base pages in a manner that is aware of the contiguity of memory allocated to each application. Figure 2 shows how a state-of-the-art GPU memory manager [81] allocates memory for two applications. Within a single *large page frame* (i.e., a contiguous piece of physical memory that is the size of a large page and whose starting address is page aligned), the GPU memory manager allocates base pages from both Applications 1 and 2 (① in Figure 2). As a result, the memory manager *cannot* coalesce the base pages into a large page (②) without first migrating some of the base pages, which would incur a high latency. Instead, *Mosaic* allocates physical base pages in a way that avoids the need to migrate data during coalescing (③ in Figure 3), and uses a simple coalescing mechanism to combine base pages into large pages (e.g., 2MB) and thus increase TLB reach (④ in Figure 3).

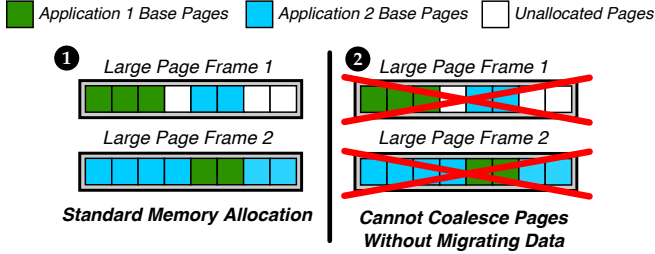


Figure 2: Page allocation and coalescing behavior of a state-of-the-art GPU memory manager [81]. Adapted from [8].

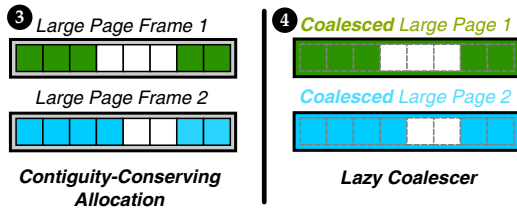


Figure 3: Page allocation and coalescing behavior of *Mosaic*. Adapted from [8].

We make a *key observation* about the memory behavior of contemporary general-purpose GPU (GPGPU) applications. We find that the vast majority of memory allocations in GPGPU applications are performed *en masse* (i.e., a large number of pages are allocated at the same time). The *en masse* memory allocation presents us with an opportunity: with so many pages being allocated at once, we can rearrange how we allocate the base pages to ensure that (1) *all* of the base pages allocated within a large page frame belong to the *same* virtual address space, and (2) base pages that are contiguous

in virtual memory are allocated to a contiguous portion of physical memory and aligned within the large page frame.

Mosaic is designed to achieve these two goals. It consists of three major components: *Contiguity-Conserving Allocation* (CoCoA), the *In-Place Coalescer*, and *Contiguity-Aware Compaction* (CAC). These three components work together to *coalesce* (i.e., combine) base pages into large pages and *splinter* (i.e., split apart) large pages back to base pages during memory management. Memory management operations for *Mosaic* take place at two times: (1) when memory is *allocated*, and (2) when memory is *deallocated*. We describe what happens at each component briefly. Figure 4 depicts the three components of *Mosaic*, and we will use Figure 4 to provide a walkthrough of the actions taken during memory allocation and deallocation.

Memory Allocation. When a GPGPU application wants to access data that is *not* currently in the GPU memory, it sends a request to the GPU runtime (e.g., OpenCL, CUDA runtimes) to transfer the data from the CPU memory to the GPU memory (① in Figure 4). A GPGPU application typically allocates a large number of base pages at the same time. CoCoA allocates space within the GPU memory (②) for the base pages, working to conserve the contiguity of base pages, if possible during allocation. Regardless of contiguity, CoCoA provides a *soft guarantee* that a single large page frame contains base pages from only a *single* application. Once the base page is allocated, CoCoA initiates the data transfer across the system I/O bus (③). When the data transfer is complete (④), CoCoA notifies the *In-Place Coalescer* that allocation is done by sending a list of the large page frame addresses that were allocated (⑤). For each of these large page frames, the runtime portion of the *In-Place Coalescer* then checks to see whether (1) *all* base pages within the large page frame have been allocated, and (2) the base pages within the large page frame are contiguous in both virtual and physical memory. If both conditions are true, the hardware portion of the *In-Place Coalescer* updates the page table to coalesce the base pages into a large page (⑥). Section 4.3 of our MICRO 2017 paper [8] describes how page tables are modified to support coalescing.

Memory Deallocation. When a GPGPU application would like to deallocate memory (e.g., when an application kernel finishes), it sends a deallocation request to the GPU runtime (⑦). For all deallocated base pages that are coalesced into a large page, the runtime invokes *Contiguity-Aware Compaction* (CAC) for the corresponding large page. The runtime portion of CAC checks to see whether the large page has a high degree of *internal fragmentation* (i.e., if the number of unallocated base pages within the large page exceeds a predetermined threshold). For each large page with high internal fragmentation, the hardware portion of CAC updates the page table to splinter the large page back into its constituent base pages (⑧). Next, CAC compacts the splintered large page frames, by migrating data from multiple splintered

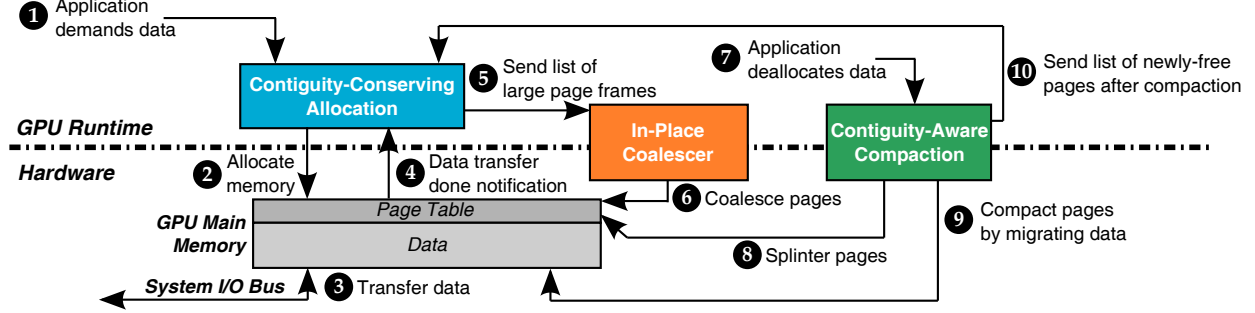


Figure 4: High-level overview of *Mosaic*, showing how and when its three components interact with the GPU memory. Reproduced from [8].

large page frames into a single large page frame (9). Finally, CAC notifies CoCoA of the large page frames that are now free after compaction (10), which CoCoA can use for future memory allocations. We describe each component of *Mosaic* in more detail in Sections 4.2, 4.3, and 4.4 of our MICRO 2017 paper [8].

3. Evaluation Methodology

Table 1 shows the system configuration we simulate for our evaluations, including the configurations of the GPU cores and memory partitions. We modify the MAFIA framework [43], which uses GPGPU-Sim 3.2.2 [10], to evaluate *Mosaic*. We add a memory allocator into *cuda-sim*, the CUDA simulator within GPGPU-Sim, to handle all virtual-to-physical address translations and to provide memory protection. We add an accurate model of address translation to GPGPU-Sim, including TLBs, page tables, and a page table walker. The page table walker is shared across all SMs, and allows up to 64 concurrent walks. Both the L1 and L2 TLBs have separate entries for base pages and large pages [32, 47, 48, 75, 78, 79]. Each TLB contains miss status holding registers (MSHRs) [54] to track in-flight page table walks. Our simulation infrastructure supports demand paging by detecting page faults and faithfully modeling the system I/O bus (i.e., PCIe) latency based on measurements from NVIDIA GTX 1080 cards [74]. We use a worst-case model for the performance of our compaction mechanism conservatively, by stalling the *entire* GPU (all SMs) and flushing the pipeline. We have publicly released our simulator modifications as open source software [88, 89].

We evaluate the performance of *Mosaic* using both *homogeneous* and *heterogeneous* workloads. We categorize each workload based on the number of concurrently-executing applications, which ranges from one to five for our homogeneous workloads, and from two to five for our heterogeneous workloads. We form our homogeneous workloads using multiple copies of the same application. We build 27 homogeneous workloads for each category using GPGPU applications from the Parboil [92], SHOC [25], LULESH [49, 50], Rodinia [20], and CUDA SDK [69] suites. We form our heterogeneous workloads by randomly selecting a number of applications out of these 27 GPGPU applications. We build 25

GPU Core Configuration	
Shader Core Config	30 cores, 1020 MHz, GTO warp scheduler [84]
Private L1 Cache	16KB, 4-way associative, LRU, L1 misses are coalesced before accessing L2, 1-cycle latency
Private L1 TLB	128 base page/16 large page entries per core, fully associative, LRU, single port, 1-cycle latency
Memory Partition Configuration	
(6 memory partitions in total with each partition accessible by all 30 cores)	
Shared L2 Cache	2MB total, 16-way associative, LRU, 2 cache banks, 2 ports per memory partition, 10-cycle latency
Shared L2 TLB	512 base page/256 large page entries, 16-way/fully-associative (base page/large page), non-inclusive, LRU, 2 ports, 10-cycle latency
DRAM	3GB GDDR5 [37, 53], 1674 MHz, 6 channels, 8 banks per rank, FR-FCFS scheduler [83, 104], burst length 8

Table 1: Configuration of the simulated system. Adapted from [8].

heterogeneous workloads per category. In total we evaluate 235 homogeneous and heterogeneous workloads.

We compare *Mosaic* to two mechanisms: (1) *GPU-MMU*, a baseline GPU with a state-of-the-art memory manager based on the work by Power et al. [81]; and (2) *Ideal TLB*, a GPU with an ideal TLB, where every address translation request hits in the L1 TLB (i.e., there are no TLB misses). We report workload performance using the weighted speedup metric [28, 29], which is calculated as:

$$\text{Weighted Speedup} = \sum \frac{IPC_{shared}}{IPC_{alone}} \quad (1)$$

where IPC_{alone} is the retired instructions per cycle (IPC) of an application in the workload that runs on the same number of shader cores using the baseline state-of-the-art configuration [81], but does *not* share GPU resources with any other applications; and IPC_{shared} is the IPC of the application when it runs concurrently with other applications. We report the performance of each application within a workload using IPC.

Section 5 of our MICRO 2017 paper [8] provides more detail on our experimental methodology.

4. Experimental Results

Figure 5 shows the performance of *Mosaic* for the homogeneous workloads we evaluated. We make two observations from the figure. First, we observe that *Mosaic* is able to recover most of the performance lost due to the overhead of address translation (i.e., an ideal TLB) in homogeneous workloads. Compared to the GPU-MMU baseline, *Mosaic* improves system performance by 55.5%, averaged across all 135 of our homogeneous workloads. The performance of *Mosaic* comes within 6.8% of the *Ideal TLB* performance, indicating that *Mosaic* is effective at extending the TLB reach. Second, we observe that *Mosaic* provides good scalability. As we increase the number of concurrently-executing applications, which puts more pressure on the shared TLBs, we observe that the performance of *Mosaic* remains close to the *Ideal TLB* performance.

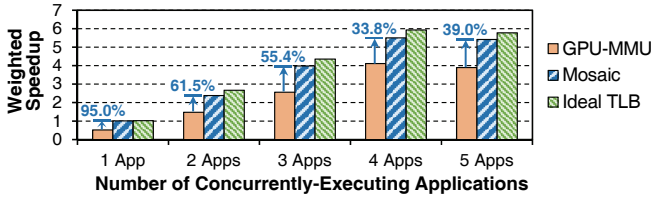


Figure 5: Homogeneous workload performance of GPU memory managers as we vary the number of concurrently-executing applications in each workload. Reproduced from [8].

Figure 6 shows the performance of *Mosaic* for heterogeneous workloads that consist of multiple different randomly-selected GPGPU applications (100 workloads in total). From the figure, we observe that on average across all of the workloads, *Mosaic* provides a performance improvement of 29.7% over GPU-MMU, and comes within 15.4% of the *Ideal TLB* performance. We find that the improvement comes from the significant reduction in the TLB miss rate with *Mosaic*. We also see that *Mosaic*'s scalability is good, as the number of applications increases, yet there is still room for improvement to reach the performance of *Ideal TLB*. We conclude that *Mosaic* is a more effective memory manager than the state-of-the-art. A detailed analysis of the results in Figures 5 and 6 can be found in Sections 6.1 and 6.2 of our MICRO 2017 paper [8].

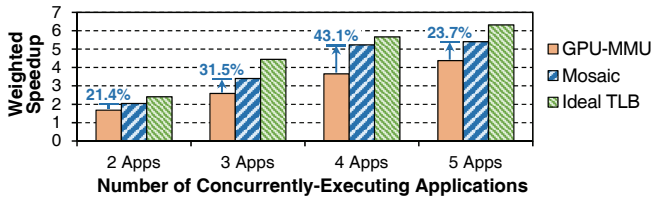


Figure 6: Heterogeneous workload performance of the GPU memory managers. Reproduced from [8].

Impact of Demand Paging on Performance. All of our results so far show the performance of the GPU-MMU

baseline and *Mosaic* when demand paging is *enabled*. Figure 7 shows the normalized weighted speedup of the GPU-MMU baseline and *Mosaic*, compared to GPU-MMU *without demand paging*, where all data required by an application is moved to the GPU memory *before* the application starts executing. We make two observations from the figure. First, we find that *Mosaic* outperforms GPU-MMU without demand paging by 58.5% on average for homogeneous workloads and 47.5% on average for heterogeneous workloads. Second, we find that demand paging has little impact on the weighted speedup. This is because demand paging latency occurs only when a kernel launches, at which point the GPU retrieves data from the CPU memory. The data transfer overhead is required regardless of whether or not demand paging is enabled, and thus the GPU incurs similar overhead with and without demand paging. We conclude that *Mosaic* improves performance significantly, regardless of the demand paging overhead in the baseline.

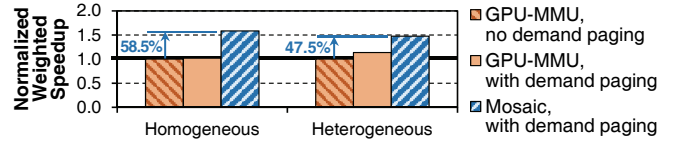


Figure 7: Performance of GPU-MMU and *Mosaic* compared to GPU-MMU without demand paging. Reproduced from [8].

TLB Hit Rate. Figure 8 compares the overall TLB hit rate of GPU-MMU to *Mosaic* for 214 of our 235 workloads, which suffer from *limited TLB reach* (i.e., workloads that have an L2 TLB hit rate lower than 98%). We make two observations from the figure. First, we observe that *Mosaic* is very effective at increasing the TLB reach of these workloads. We find that for the GPU-MMU baseline, *every* fully-mapped large page frame contains pages from *multiple* applications, as the GPU-MMU allocator does *not* provide the soft guarantee of CoCoA (i.e., a single large page frame contains base pages from only a *single* application). As a result, GPU-MMU does not have any opportunities to coalesce base pages into a large page *without* performing significant amounts of data migration. In contrast, *Mosaic* can coalesce a vast majority of base pages thanks to CoCoA. As a result, *Mosaic* reduces the TLB miss rate drastically for these workloads, with the average miss rate falling below 1% in both the L1 and L2 TLBs. Second, we observe an increasing amount of interference in GPU-MMU when more than three applications are running concurrently. This results in a lower TLB hit rate as the number of applications increases from three to four, and from four to five. The L2 TLB hit rate of GPU-MMU drops from 81% in workloads with two concurrently-executing applications to 62% in workloads with five concurrently-executing applications. *Mosaic* experiences no such drop due to interference as we increase the number of concurrently-executing applications, since it makes much greater use of large page coalescing and enables a much larger TLB reach. We conclude that *Mosaic* is very effective in improving the hit rate.

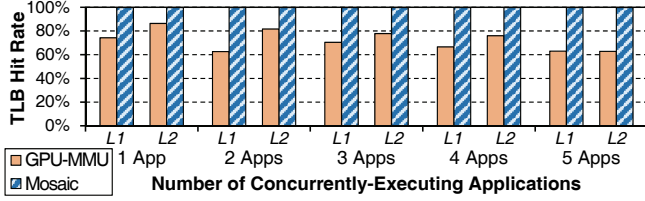


Figure 8: L1 and L2 TLB hit rates for GPU-MMU and *Mosaic*. Reproduced from [8].

We provide the following additional results in our full MICRO 2017 paper [8]:

- Individual applications’ performance with *Mosaic* and the baseline GPU-MMU
- TLB size sensitivity of *Mosaic* and the baseline GPU-MMU
- Analysis of the effectiveness of CAC to reduce memory fragmentation incurs by using large pages

5. Related Work

To our knowledge, this is the first work to (1) analyze the fundamental trade-offs between TLB reach, demand paging performance, and internal page fragmentation; and (2) propose an application-transparent GPU memory manager that preemptively coalesces pages at allocation time to improve address translation performance, while avoiding the demand paging inefficiencies and memory copy overheads typically associated with large page support. Reducing performance degradation from address translation overhead is an active area of work for CPUs, and the performance loss that we observe as a result of address translation is well corroborated [13, 15, 31, 33, 63]. In this section, we discuss previous techniques that aim to reduce the overhead of address translation and demand paging.

5.1. TLB Designs for CPU Systems

TLB miss overhead can be reduced by (1) accelerating page table walks [11, 14] or reducing the walk frequency [32]; or (2) reducing the number of TLB misses (e.g., through prefetching [16, 46, 90], prediction [75], structural changes to the TLB [77, 78, 93] or a TLB hierarchy [4, 5, 13, 15, 31, 47, 60, 91]).

Support for Multiple Page Sizes. Multi-page mapping techniques [77, 78, 93] use a single TLB entry for multiple page translations, improving TLB reach by a small factor. Much greater improvements to TLB reach are needed to deal with modern memory sizes. MIX TLB [24] accommodates entries that translate multiple page sizes, eliminating the need for a dedicated set of large page entries in the TLB. MIX TLB is orthogonal to our work, and can be used with *Mosaic* to further improve TLB reach.

Navarro et al. [66] identify contiguity-awareness and fragmentation reduction as primary concerns for large page management, proposing reservation-based allocation and deferred promotion (i.e., coalescing) of base pages to large pages. Similar ideas are widely used in modern OSes [23]. Instead of the

reservation-based scheme, Ingens [56] employs a utilization-based scheme that uses a bit vector to track spatial and temporal utilization of base pages.

Techniques to Increase Memory Contiguity.

GLUE [79] groups contiguous, aligned base page translations under a single speculative large page translation in the TLB. GTSM [26] provides hardware support to leverage the contiguity of physical memory region even when pages have been retired due to bit errors. These mechanisms for preserving or recovering contiguity are orthogonal to the contiguity-conserving allocation we propose for *Mosaic*, and they can help *Mosaic* by avoiding the need for compaction.

Gorman et al. [35] propose a placement policy for an OS’s physical page allocator that mitigates fragmentation and promotes contiguity by grouping pages according to the amount of migration required to achieve contiguity. Subsequent work [36] proposes a software-exposed interface for applications to explicitly request large pages like `libhugetlbfs` [34]. These ideas are complementary to our work. *Mosaic* can potentially benefit from similar policies if such policies can be simplified enough to be implementable in hardware.

Alternative TLB Designs. Research on shared last-level TLB designs [15, 17, 60] and page walk cache designs [14] has yielded mechanisms that accelerate multithreaded CPU applications by sharing translations between cores. SpecTLB [12] provides a technique that predicts address translations. While speculation works on CPU applications, speculation for highly-parallel GPUs is more complicated [41, 44], and can potentially waste off-chip DRAM bandwidth, which is a highly-contended resource in GPUs. Direct segments [13] and redundant memory mappings [47] provide virtual memory support for server workloads that reduces the overhead of address translation. These proposals map large contiguous chunks of virtual memory to the physical address space in order to reduce the address translation overhead. While these techniques improve the TLB reach, they increase the transfer latency depending on the size of the virtual chunks they map.

5.2. TLB Designs for GPU Systems

TLB Designs for Heterogeneous Systems. Previous works provide several TLB designs for heterogeneous systems with GPUs [80, 81, 95] and with accelerators [22]. *Mosaic* improves upon a state-of-the-art TLB design [81] by providing application-transparent, high-performance support for multiple page sizes in GPUs. No prior work provides such support.

TLB-Aware Warp Scheduler. Pichai et al. [80] extend the cache-conscious warp scheduler [84] to be aware of the TLB in heterogeneous CPU-GPU systems. Other more sophisticated warp schedulers [51, 59, 62, 65, 84, 85, 103] can also be extended to be TLB aware. These techniques are orthogonal to the problem we focus on, and can be applied in conjunction with *Mosaic* to further improve performance.

TLB-Aware Memory Hierarchy. Ausavarungnirun et al. [9] improve the performance of the GPU under the presence of memory protection by redesigning the GPU main memory hierarchy to be aware of TLB-related memory requests. Many prior works propose memory scheduling designs for GPUs [7, 42, 45, 101] and heterogeneous systems [6, 94]. These memory scheduling design can be modified to be aware of TLB-related memory requests and used in conjunction with *Mosaic* to further improve the performance of the GPUs.

Analysis of Address Translation in GPUs. Vesely et al. [95] analyze support for virtual memory in heterogeneous systems, finding that the cost of address translation in GPUs is an order of magnitude higher than that in CPUs. They observe that high-latency address translations limit the GPU’s latency hiding capability and hurt performance. Mei et al. [61] use a set of microbenchmarks to evaluate the address translation process in commercial GPUs. Their work concludes that previous NVIDIA architectures [71, 72] have *off-chip* L1 and L2 TLBs, which lead to poor performance.

GPU Core Modifications. Many prior works propose modifications to the GPU core design [7, 45, 51, 52, 55, 59, 62, 65, 84, 85, 86, 87, 97, 98, 103]. These techniques are complementary to *Mosaic*, and can be combined with *Mosaic* to further improve GPU performance.

5.3. GPU Virtualization

VAST [58] is a software-managed virtual memory space for GPUs. In that work, the authors observe that the limited size of physical memory typically prevents data-parallel programs from utilizing GPUs. To address this, VAST automatically partitions GPU programs into chunks that fit within the physical memory space to create the illusion of infinite virtual memory. Unlike *Mosaic*, VAST is unable to provide memory protection from concurrently-executing GPGPU applications. Zorua [96] is a holistic mechanism to virtualize multiple hardware resources within the GPU. Zorua does not virtualize the main memory, and is thus orthogonal to our work. vmCUDA [99] and rCUDA [27] provide close-to-ideal performance, but they require significant modifications to GPGPU applications and the operating system, which sacrifice transparency to the application, performance isolation, and compatibility across multiple GPU architectures.

5.4. Demand Paging for GPUs

Demand paging is a challenge for GPUs [95]. Recent works [3, 102], and the AMD hUMA [57] and NVIDIA Pascal architectures [73, 102] provide various levels of support for demand paging in GPUs. These techniques do *not* tackle the existing trade-off in GPUs between using large pages to improve address translation and using base pages to minimize demand paging overhead, which we relax with *Mosaic*.

6. Potential Impact of Mosaic

While several previous works propose mechanisms to lower the overhead of virtual memory [13, 15, 26, 31, 33, 63, 79,

80, 81, 95], only a handful of these works extensively evaluate virtual memory on GPUs [58, 80, 81, 95], and no work has investigated virtual memory as a shared resource when *multiple* GPGPU applications need to share the GPUs. In this section, we explore the potential future impact of *Mosaic*.

Support for Concurrent Application Execution in GPUs. The large number of cores within a contemporary GPU make it an attractive substrate for executing multiple applications in parallel. This can be especially useful in virtualized cloud environments, where hardware resources are safely partitioned across multiple virtual machines to provide efficient resource sharing. Prior approaches to execute multiple applications concurrently on a GPU have been limited, as they either (1) lack sufficient memory protection support across multiple applications; (2) incur a high performance overhead to provide memory protection; or (3) perform a conservative static partitioning of the GPU, which can often underutilize many resources in the GPU.

Mosaic provides the first flexible support for memory protection within a GPU, allowing applications to dynamically partition GPU resources *without* violating memory protection guarantees. This support can enable the practical virtualization and sharing of GPUs in a cloud environment, which in turn can increase the appeal of GPGPU programming and the use cases of GPGPUs. By enabling practical support for concurrent application execution on GPUs, *Mosaic* encourages and enables future research in several areas, such as resource sharing mechanisms, kernel scheduling, and quality-of-service enforcement within the GPU and heterogeneous systems.

Virtual Memory for SIMD Architectures. *Mosaic* is an important first step to enable *low overhead virtual memory* in GPUs. We believe that the key ideas and general observations that we make are applicable to any highly-parallel SIMD architecture [30], and to heterogeneous systems with SIMD-based processing cores [1, 18, 19, 21, 38, 39, 40, 64, 67, 68, 82, 100]. Future works can expand upon our findings and adapt our mechanisms to reduce the overhead of page walks and demand paging on other SIMD-based systems.

Improved Programmability. Aside from memory protection, virtual memory can be used to (1) improve the programmability of GPGPU applications, and (2) decouple a GPU kernel’s working set size from the size of the GPU memory. In fact, *Mosaic* transparently allows applications to benefit from virtual memory without incurring a significant performance overhead. This is a key advantage for programmers, many of whom are used to the conventional programming model used in CPUs to provide application portability and memory protection. By providing programmers with a familiar and simple memory abstraction, we expect that a greater number of programmers will start writing high-performance GPGPU applications. Furthermore, by enabling low-overhead memory virtualization, *Mosaic* can enable new classes of GPGPU applications. For example, in the past, programmers were not able

to easily write GPGPU applications whose memory working set sizes exceeded the physical memory within the GPU. With *Mosaic*, programmers no longer need to restrict themselves to applications whose working sets fit within the physical memory; they can rely on the GPU itself software-transparently managing page migration and address translation.

Publicly-Released Infrastructure. Our simulation infrastructure is publicly available as open-source software [88]. Other researchers can utilize our infrastructure to conduct future research on virtual memory management on GPUs. Some examples of research topics that can be investigated using our infrastructure include (1) how to manage which pages reside in CPU memory or GPU memory, (2) how to dynamically partition the physical main memory across multiple concurrently-executing applications, and (3) how to maintain programmability of the virtual memory as the GPU architecture evolves and becomes more heterogeneous over time. We hope and believe that our new, open-source infrastructure can inspire future research in these and other research areas on GPU and heterogeneous system memory virtualization.

7. Conclusion

We introduce *Mosaic*, a new GPU memory manager that provides application-transparent support for multiple page sizes. The key idea of *Mosaic* is to perform demand paging using *smaller* page sizes, and then coalesce small (i.e., base) pages into a *larger* page immediately after allocation, which allows address translation to use large pages and thus increase TLB reach. We have shown that *Mosaic* significantly outperforms state-of-the-art GPU address translation designs and achieves performance close to an ideal TLB, across a wide variety of workloads. We conclude that *Mosaic* effectively combines the benefits of large pages and demand paging in GPUs, thereby breaking the conventional tension that exists between these two concepts. We hope the ideas presented in our MICRO 2017 paper can lead to future works that analyze *Mosaic* in detail and provide even lower-overhead support for synergistic address translation and demand paging in GPUs and heterogeneous systems.

Acknowledgments

We thank the anonymous reviewers and SAFARI group members for their feedback. Special thanks to Nastaran Hajinazar, Juan Gómez Luna, and Mohammad Sadr for their feedback. We acknowledge the support of our industrial partners, especially Google, Intel, Microsoft, NVIDIA, Samsung, and VMware. This research was partially supported by the NSF (grants 1409723 and 1618563), the Intel Science and Technology Center for Cloud Computing, and the Semiconductor Research Corporation.

References

- [1] Advanced Micro Devices, Inc., “AMD Accelerated Processing Units,” <http://www.amd.com/us/products/technologies/apu/Pages/apu.aspx>.
- [2] Advanced Micro Devices, Inc., “OpenCL: The Future of Accelerated Application Performance Is Now,” https://www.amd.com/Documents/FirePro_OpenCL_Whitepaper.pdf.
- [3] N. Agarwal, D. Nellans, M. O’Connor, S. W. Keckler, and T. F. Wenisch, “Unlocking Bandwidth for GPUs in CC-NUMA Systems,” in *HPCA*, 2015.
- [4] J. Ahn, S. Jin, and J. Huh, “Revisiting Hardware-Assisted Page Walks for Virtualized Systems,” in *ISCA*, 2012.
- [5] J. Ahn, S. Jin, and J. Huh, “Fast Two-Level Address Translation for Virtualized Systems,” *IEEE TC*, 2015.
- [6] R. Ausavarungnirun, K. Chang, L. Subramanian, G. Loh, and O. Mutlu, “Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems,” in *ISCA*, 2012.
- [7] R. Ausavarungnirun, S. Ghose, O. Kayiran, G. H. Loh, C. R. Das, M. T. Kandemir, and O. Mutlu, “Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance,” in *PACT*, 2015.
- [8] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu, “Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes,” in *MICRO*, 2017.
- [9] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. Rossbach, and O. Mutlu, “MASK: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency,” in *ASPLOS*, 2018.
- [10] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, “Analyzing CUDA Workloads Using a Detailed GPU Simulator,” in *ISPASS*, 2009.
- [11] T. W. Barr, A. L. Cox, and S. Rixner, “Translation Caching: Skip, Don’t Walk (the Page Table),” in *ISCA*, 2010.
- [12] T. W. Barr, A. L. Cox, and S. Rixner, “SpecTLB: A Mechanism for Speculative Address Translation,” in *ISCA*, 2011.
- [13] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, “Efficient Virtual Memory for Big Memory Servers,” in *ISCA*, 2013.
- [14] A. Bhattacharjee, “Large-Reach Memory Management Unit Caches,” in *MICRO*, 2013.
- [15] A. Bhattacharjee, D. Lustig, and M. Martonosi, “Shared Last-level TLBs for Chip Multiprocessors,” in *HPCA*, 2011.
- [16] A. Bhattacharjee and M. Martonosi, “Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors,” in *PACT*, 2009.
- [17] A. Bhattacharjee and M. Martonosi, “Inter-Core Cooperative TLB for Chip Multiprocessors,” in *ASPLOS*, 2010.
- [18] D. Bouvier and B. Sander, “Applying AMD’s ‘Kaveri’ APU for Heterogeneous Computing,” in *Hot Chips*, 2014.
- [19] B. Burgess, B. Cohen, J. Dundas, J. Rupley, D. Kaplan, and M. Denman, “Bobcat: AMD’s Low-Power x86 Processor,” *IEEE Micro*, 2011.
- [20] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, “Rodia: A Benchmark Suite for Heterogeneous Computing,” in *IISWC*, 2009.
- [21] M. Clark, “A New X86 Core Architecture for the Next Generation of Computing,” in *Hot Chips*, 2016.
- [22] J. Cong, Z. Fang, Y. Hao, and G. Reinman, “Supporting Address Translation for Accelerator-Centric Architectures,” in *HPCA*, 2017.
- [23] J. Corbet, “Transparent Hugepages,” <https://lwn.net/Articles/359158/>, 2009.
- [24] G. Cox and A. Bhattacharjee, “Efficient Address Translation for Architectures with Multiple Page Sizes,” in *ASPLOS*, 2017.
- [25] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, “The Scalable Heterogeneous Computing (SHOC) Benchmark Suite,” in *GPGPU*, 2010.
- [26] Y. Du, M. Zhou, B. R. Childers, D. Mossé, and R. Melhem, “Supporting Superpages in Non-Contiguous Physical Memory,” in *HPCA*, 2015.
- [27] J. Duato, A. Pena, F. Silla, R. Mayo, and E. Quintana-Orti, “rCUDA: Reducing the Number of GPU-based Accelerators in High Performance Clusters,” in *HPSC*, 2010.
- [28] S. Eyerman and L. Eeckhout, “System-Level Performance Metrics for Multiprogram Workloads,” *IEEE Micro*, 2008.
- [29] S. Eyerman and L. Eeckhout, “Restating the Case for Weighted-IPC Metrics to Evaluate Multiprogram Workload Performance,” *IEEE CAL*, 2014.
- [30] M. Flynn, “Very High-Speed Computing Systems,” *Proc. of the IEEE*, vol. 54, no. 2, 1966.
- [31] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift, “Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks,” in *MICRO*, 2014.
- [32] J. Gandhi, M. D. Hill, and M. M. Swift, “Agile Paging: Exceeding the Best of Nested and Shadow Paging,” in *ISCA*, 2016.
- [33] F. Gaud, B. Lepers, J. Decouchant, J. Funston, A. Fedorova, and V. Quema, “Large Pages May Be Harmful on NUMA Systems,” in *USENIX ATC*, 2014.
- [34] M. Gorman, “Huge Pages Part 2 (Interfaces),” <https://lwn.net/Articles/375096/>, 2010.
- [35] M. Gorman and P. Healy, “Supporting Superpage Allocation Without Additional Hardware Support,” in *ISMM*, 2008.
- [36] M. Gorman and P. Healy, “Performance Characteristics of Explicit Superpage Support,” in *WIOSCA*, 2010.
- [37] Hynix. Hynix GDDR5 SGRAM Part H5GQ1H24AFR Revision 1.0. [http://www.hynix.com/datasheet/pdf/graphics/H5GQ1H24AFR\(Rev1.0\).pdf](http://www.hynix.com/datasheet/pdf/graphics/H5GQ1H24AFR(Rev1.0).pdf)
- [38] Intel Corp., “Sandy Bridge Intel Processor Graphics Performance Developer’s Guide.”
- [39] Intel Corp., “Introduction to Intel® Architecture,” 2014.

- [40] Intel Corp., “6th Generation Intel® Core™ Processor Family Datasheet, Vol. 1,” 2017.
- [41] J. A. Jablin, T. B. Jablin, O. Mutlu, and M. Herlihy, “Warp-aware Trace Scheduling for GPUs,” in *PACT*, 2014.
- [42] M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver, “A QoS-Aware Memory Controller for Dynamically Balancing GPU and CPU Bandwidth Use in an MPSoC,” in *DAC*, 2012.
- [43] A. Jog, O. Kayiran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S. W. Keckler, M. T. Kandemir, and C. R. Das, “Anatomy of GPU Memory System for Multi-Application Execution,” in *MEMSYS*, 2015.
- [44] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, “Orchestrated Scheduling and Prefetching for GPGPUs,” in *ISCA*, 2013.
- [45] A. Jog, O. Kayiran, A. Pattnaik, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, “Exploiting Core Criticality for Enhanced GPU Performance,” in *SIGMETRICS*, 2016.
- [46] G. B. Kandaraju and A. Sivasubramaniam, “Going the Distance for TLB Prefetching: An Application-Driven Study,” in *ISCA*, 2002.
- [47] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, “Redundant Memory Mappings for Fast Access to Large Memories,” in *ISCA*, 2015.
- [48] V. Karakostas, J. Gandhi, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, “Energy-Efficient Address Translation,” in *HPCA*, 2016.
- [49] I. Karlin *et al.*, “Exploring Traditional and Emerging Parallel Programming Models Using a Proxy Application,” in *IPDPS*, 2013.
- [50] I. Karlin, J. Keasler, and R. Neely, “LULESH 2.0 Updates and Changes,” Lawrence Livermore National Lab, Tech. Rep. LLNL-TR-641973, 2013.
- [51] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das, “Neither More Nor Less: Optimizing Thread-Level Parallelism for GPGPUs,” in *PACT*, 2013.
- [52] O. Kayiran, N. C. Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das, “Managing GPU Concurrency in Heterogeneous Architectures,” in *MICRO*, 2014.
- [53] Y. Kim, W. Yang, and O. Mutlu, “Ramulator: A Fast and Extensible DRAM Simulator,” *CAL*, 2015.
- [54] D. Kroft, “Lockup-Free Instruction Fetch/Prefetch Cache Organization,” in *ISCA*, 1981.
- [55] H.-K. Kuo, B. C. Lai, and J.-Y. Jou, “Reducing Contention in Shared Last-Level Cache for Throughput Processors,” *ACM TODAES*, vol. 20, no. 1, 2014.
- [56] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel, “Coordinated and Efficient Huge Page Management with Ingens,” in *OSDI*, 2016.
- [57] G. Kyriazis, “Heterogeneous System Architecture: A Technical Review,” <https://developer.amd.com/wordpress/media/2012/10/hsa10.pdf>, Advanced Micro Devices, Inc., 2012.
- [58] J. Lee, M. Samadi, and S. Mahlke, “VAST: The Illusion of a Large Memory Space for GPUs,” in *PACT*, 2014.
- [59] S.-Y. Lee and C.-J. Wu, “CAWS: Criticality-Aware Warp Scheduling for GPGPU Workloads,” in *PACT*, 2014.
- [60] D. Lustig, A. Bhattacharjee, and M. Martonosi, “TLB Improvements for Chip Multiprocessors: Inter-Core Cooperative Prefetchers and Shared Last-Level TLBs,” *ACM TACO*, 2013.
- [61] X. Mei and X. Chu, “Dissecting GPU Memory Hierarchy Through Microbenchmarking,” *IEEE TPDS*, 2017.
- [62] J. Meng, D. Tarjan, and K. Skadron, “Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance,” in *ISCA*, 2010.
- [63] T. Merrifield and H. R. Taheri, “Performance Implications of Extended Page Tables on Virtualized x86 Processors,” in *VEE*, 2016.
- [64] R. Mijat, “Take GPU Processing Power Beyond Graphics with Mali GPU Computing,” 2012.
- [65] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, “Improving GPU Performance via Large Warps and Two-Level Warp Scheduling,” in *MICRO*, 2011.
- [66] J. Navarro, S. Iyer, P. Druschel, and A. Cox, “Practical, Transparent Operating System Support for Superpages,” in *OSDI*, 2002.
- [67] NVIDIA Corp., “NVIDIA Tegra K1,” http://www.nvidia.com/content/pdf/tegra_white_papers/tegra-k1-whitepaper-v1.0.pdf.
- [68] NVIDIA Corp., “NVIDIA Tegra X1,” <https://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf>.
- [69] NVIDIA Corp., “CUDA C/C++ SDK Code Samples,” 2011.
- [70] NVIDIA Corp., “NVIDIA’s Next Generation CUDA Compute Architecture: Fermi,” http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf, 2011.
- [71] NVIDIA Corp., “NVIDIA’s Next Generation CUDA Compute Architecture: Kepler GK110,” 2012.
- [72] NVIDIA Corp., “NVIDIA GeForce GTX 750 Ti,” 2014.
- [73] NVIDIA Corp., “NVIDIA Tesla P100,” 2016.
- [74] NVIDIA Corp., “NVIDIA GeForce GTX 1080,” 2017.
- [75] M.-M. Papadopolou, X. Tong, A. Sez nec, and A. Moshovos, “Prediction-Based Superpage-Friendly TLB Designs,” in *HPCA*, 2015.
- [76] PCI-SIG, “PCI Express Base Specification Revision 3.1a,” 2015.
- [77] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, “Increasing TLB Reach by Exploiting Clustering in Page Translations,” in *HPCA*, 2014.
- [78] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, “CoLT: Coalesced Large-Reach TLBs,” in *MICRO*, 2012.
- [79] B. Pham, J. Vesely, G. Loh, and A. Bhattacharjee, “Large Pages and Lightweight Memory Management in Virtualized Systems: Can You Have It Both Ways?” in *MICRO*, 2015.
- [80] B. Pichai, L. Hsu, and A. Bhattacharjee, “Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces,” in *ASPLOS*, 2014.
- [81] J. Power, M. D. Hill, and D. A. Wood, “Supporting x86-64 Address Translation for 100s of GPU Lanes,” in *HPCA*, 2014.
- [82] PowerVR, “PowerVR Hardware Architecture Overview for Developers,” <http://cdn.imgtec.com/sdk-documentation/PowerVR+Hardware+Architecture+Overview+for+Developers.pdf>, 2016.
- [83] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, “Memory Access Scheduling,” in *ISCA*, 2000.
- [84] T. G. Rogers, M. O’Connor, and T. M. Aamodt, “Cache-Conscious Wavefront Scheduling,” in *MICRO*, 2012.
- [85] T. G. Rogers, M. O’Connor, and T. M. Aamodt, “Divergence-Aware Warp Scheduling,” in *MICRO*, 2013.
- [86] M. Sadrosadati, A. Mirhosseini, B. Ehsani, H. Sarbazi-Azad, M. P. Drumond, B. Falsafi, R. Ausavarungnirun, and O. Mutlu, “LTRF: A Latency Tolerant Register File Architecture for GPUs,” in *ASPLOS*, 2018.
- [87] M. Sadrosadati, A. Mirhosseini, B. Ehsani, H. Sarbazi-Azad, M. P. Drumond, B. Falsafi, R. Ausavarungnirun, and O. Mutlu, “LTRF: Enabling High-Capacity Register Files for GPUs via Hardware/Software Cooperative Register Prefetching,” in *ASPLOS*, 2018.
- [88] SAFARI Research Group, “Mosaic – GitHub Repository,” <https://github.com/CMU-SAFARI/Mosaic/>.
- [89] SAFARI Research Group, “SAFARI Software Tools – GitHub Repository,” <https://github.com/CMU-SAFARI/>.
- [90] A. Saulsbury, F. Dahlgren, and P. Stenström, “Recency-Based TLB Preloading,” in *ISCA*, 2000.
- [91] S. Srikanthiah and M. Kandemir, “Synergistic TLBs for High Performance Address Translation in Chip Multiprocessors,” in *MICRO*, 2010.
- [92] J. A. Stratton, C. Rodrigues, I. J. Sung, N. Obeid, L. W. Chang, N. Anssari, G. D. Liu, and W. W. Hwu, “Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing,” Univ. of Illinois at Urbana-Champaign, IMPACT Research Group, Tech. Rep. IMPACT-12-01, 2012.
- [93] M. Talluri and M. D. Hill, “Surpassing the TLB Performance of Superpages with Less Operating System Support,” in *ASPLOS*, 1994.
- [94] H. Usui, L. Subramanian, K. Chang, and O. Mutlu, “DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators,” *ACM TACO*, vol. 12, no. 4, Jan. 2016.
- [95] J. Vesely, A. Basu, M. Oskin, G. H. Loh, and A. Bhattacharjee, “Observations and Opportunities in Architecting Shared Virtual Memory for Heterogeneous Systems,” in *ISPASS*, 2016.
- [96] N. Vijaykumar, K. Hsieh, G. Pekhimenko, S. Khan, A. Shrestha, S. Ghose, A. Jog, P. B. Gibbons, and O. Mutlu, “Zorua: A Holistic Approach to Resource Virtualization in GPUs,” in *MICRO*, 2016.
- [97] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungnirun, C. Das, M. Kandemir, T. C. Mowry, and O. Mutlu, “A Case for Core-Assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps,” in *ISCA*, 2015.
- [98] N. Vijaykumar, G. Pekhimenko, A. Jog, S. Ghose, A. Bhowmick, R. Ausavarungnirun, C. Das, M. Kandemir, T. C. Mowry, and O. Mutlu, “A Framework for Accelerating Bottlenecks in GPU Execution with Assist Warps,” in *Advances in GPU Research and Practice*, 2016.
- [99] L. Vu, H. Sivaraman, and R. Bidarkar, “GPU Virtualization for High Performance General Purpose Computing on the ESX Hypervisor,” in *HPC*, 2014.
- [100] S. Wasson, “AMD’s A8-3800 Fusion APU,” <http://techreport.com/articles/x/21730>, 2011.
- [101] G. Yuan, A. Bakhoda, and T. Aamodt, “Complexity Effective Memory Access Scheduling for Many-Core Accelerator Architectures,” in *MICRO*, 2009.
- [102] T. Zheng, D. Nellans, A. Zulficar, M. Stephenson, and S. W. Keckler, “Towards High Performance Paged Memory for GPUs,” in *HPCA*, 2016.
- [103] Z. Zheng, Z. Wang, and M. Lipasti, “Adaptive Cache and Concurrency Allocation on GPGPUs,” *IEEE CAL*, 2014.
- [104] W. K. Zuravleff and T. Robinson, “Controller for a Synchronous DRAM That Maximizes Throughput by Allowing Memory Requests and Commands to Be Issued Out of Order,” US Patent No. 5,630,096, 1997.

Reviewers:

Australia

Abramov, Vyacheslav; Monash University
Begg, Rezau; Victoria University
Bem, Derek; University of Western Sydney
Betts, Christopher; Pegacat Computing Pty. Ltd.
Buyya, Rajkumar; The University of Melbourne
Chapman, Judith; Australian University Limited
Chen, Yi-Ping Phoebe; Deakin University
Hammond, Mark; Flinders University
Henman, Paul; University of Queensland
Palmisano, Stephen; University of Wollongong
Ristic, Branko; Science and Technology Organisation
Sajjanhar, Atul; Deakin University
Sidhu, Amandeep; University of Technology, Sydney
Sudweeks, Fay; Murdoch University

Austria

Derntl, Michael; University of Vienna
Hug, Theo; University of Innsbruck
Loidl, Susanne; Johannes Kepler University Linz
Stockinger, Heinz; University of Vienna
Sutter, Matthias; University of Innsbruck

Brazil

Parracho, Annibal; Universidade Federal Fluminense
Traina, Agma; University of Sao Paulo
Traina, Caetano; University of Sao Paulo
Vicari, Rosa; Federal University of Rio Grande

Belgium

Huang, Ping; European Commission

Canada

Fung, Benjamin; Simon Fraser University
Grayson, Paul; York University
Gray, Bette; Alberta Education
Memmi, Daniel; UQAM
Neti, Sangeeta; University of Victoria
Nickull, Duane; Adobe Systems, Inc.
Ollivier-Gooch, Carl; The University of British Columbia
Paulin, Michele; Concordia University
Plaisent, Michel; University of Quebec
Reid, Keith; Ontario Ministry of Agriculture
Shewchenko, Nicholas; Biokinetics and Associates
Steffan, Gregory; University of Toronto
Vandenbergh, Christian; HEC Montreal

Czech Republic

Kala, Zdenek; Brno University of Technology
Korab, Vojtech; Brno University of Technology
Lhotska, Lenka; Czech Technical University

Finland

Lahdelma, Risto; University of Turku
Salminen, Pekka; University of Jyväskylä

France

Cardey, Sylviane; University of Franche-Comte
Klinger, Evelyne; LTCI – ENST, Paris
Roche, Christophe; University of Savoie
Valette, Robert; LAAS - CNRS

Germany

Accorsi, Rafael; University of Freiburg
Glatzer, Wolfgang; Goethe-University
Gradmann, Stefan; Universität Hamburg
Groll, Andre; University of Siegen
Klamma, Ralf; RWTH Aachen University
Wurtz, Rolf P.; Ruhr-Universität Bochum

India

Pareek, Deepak; Technology4Development
Scaria, Vinod; Institute of Integrative Biology
Shah, Mugdha; Mansukhlal Svayam

Ireland

Eisenberg, Jacob; University College Dublin

Israel

Feintuch, Uri; Hadassah-Hebrew University

Italy

Badia, Leonardo; IMT Institute for Advanced Studies
Berritella, Maria; University of Palermo
Carpaneto, Enrico; Politecnico di Torino

Japan

Hattori, Yasunao; Shimane University
Livingston, Paisley; Lingnan University
Srinivas, Hari; Global Development Research Center

Obayashi, Shigeru; Institute of Fluid Science, Tohoku University

Netherlands

Mills, Melinda C.; University of Groningen
Pires, Luis Ferreira; University of Twente

New Zealand

Anderson, Tim; Van Der Veer Institute

Portugal

Cardoso, Jorge; University of Madeira
Natividade, Eduardo; Polytechnic Institute of Coimbra
Oliveira, Eugenio; University of Porto

Singapore

Tan, Fock-Lai; Nanyang Technological University

South Korea

Kwon, Wook Hyun; Seoul National University

Spain

Barrera, Juan Pablo Soto; University of Castilla
Gonzalez, Evelio J.; University of La Laguna
Perez, Juan Mendez; Universidad de La Laguna
Royuela, Vicente; Universidad de Barcelona
Vizcaino, Aurora; University of Castilla-La Mancha
Vilarrasa, Clelia Colombo; Open University of Catalonia

Sweden

Johansson, Mats; Royal Institute of Technology

Switzerland

Niinimäki, Marko; Helsinki Institute of Physics
Pletka, Roman; AdNovum Informatik AG
Rizzotti, Sven; University of Basel
Specht, Matthias; University of Zurich

Taiwan

Lin, Hsiung Cheng; Chienkuo Technology University
Shyu, Yuh-Huei; Tamkang University
Sue, Chuan-Ching; National Cheng Kung University

United Kingdom

Ariwa, Ezendu; London Metropolitan University
Biggam, John; Glasgow Caledonian University
Coleman, Shirley; University of Newcastle
Conole, Grainne; University of Southampton
Dorfler, Viktor; Strathclyde University
Engelmann, Dirk; University of London
Eze, Emmanuel; University of Hull
Forrester, John; Stockholm Environment Institute
Jensen, Jens; STFC Rutherford Appleton Laboratory
Kolovos, Dimitrios S.; The University of York
McBurney, Peter; University of Liverpool
Vetta, Atam; Oxford Brookes University
WHYTE, William Stewart; University of Leeds
Xie, Changwen; Wicks and Wilson Limited

USA

Bach, Eric; University of Wisconsin
Bolzendahl, Catherine; University of California
Bussler, Christoph; Cisco Systems, Inc.
Charpentier, Michel; University of New Hampshire
Chong, Stephen; Cornell University
Collison, George; The Concord Consortium
DeWeaver, Eric; University of Wisconsin - Madison
Gans, Eric; University of California
Gill, Sam; San Francisco State University
Hunter, Lynette; University of California Davis
Iceland, John; University of Maryland
Kaplan, Samantha W.; University of Wisconsin
Langou, Julien; The University of Tennessee
Liu, Yuliang; Southern Illinois University Edwardsville
Lok, Benjamin; University of Florida
Minh, Chi Cao; Stanford University
Morrissey, Robert; The University of Chicago
Mui, Lik; Google, Inc
Rizzo, Albert; University of Southern California
Rosenberg, Jonathan M.; University of Maryland
Shaffer, Cliff; Virginia Tech
Sherman, Elaine; Hofstra University
Snyder, David F.; Texas State University
Song, Zhe; University of Iowa
Wei, Chen; Intelligent Automation, Inc.
Yu, Zhiyi; University of California

Authors of papers are responsible for the contents and layout of their papers.

Welcome to IPSI BgD Conferences and Journals!

<http://tar.ipsitransactions.org>

<http://www.ipsitransactions.org>

**CIP – Katalogizacija u publikaciji
Narodna biblioteka Srbije, Beograd**

ISSN 1820 – 4511

**The IPSI BGD Transactions
on Advanced Research**

COBISS.SR - ID 119128844

