2018

IMPROVING THE PERFORMANCE AND ENERGY EFFICIENCY OF EMERGING MEMORY SYSTEMS

Yuhua Guo
Virginia Commonwealth University

IMPROVING THE PERFORMANCE AND ENERGY EFFICIENCY OF EMERGING MEMORY SYSTEMS

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at Virginia Commonwealth University.

by

YUHUA GUO

M.S., Huazhong Univ. of Sci. and Tech. - Sep. 2008 to Apr. 2011

B.S., Shaanxi Univ. of Sci. and Tech. - Sep. 2003 to Jul. 2007

Advisor: Weijun Xiao, Ph.D.,

Assistant Professor, Department of Electrical and Computer Engineering

Virginia Commonwealth University

Co-advisor: Xubin He, Ph.D.,

Professor, Department of Computer and Information Sciences

Temple University

Virginia Commonwealth University

Richmond, Virginia

May, 2018

Acknowledgements

I would like to thank my advisor Dr. Weijun Xiao and co-advisor Dr. Xubin He, for their training and guidance on my Ph.D. study and research. I give my special thanks to Dr. He, for his patience and advice on my research, encouraging me to attend academic conferences, introducing intern opportunity for me, and his generous support to help me finish my Ph.D. program. I would like to express my gratitude to Dr. Qing Liu as my mentor at Oak Ridge National Laboratory. He helped to polish my first paper and made it accepted.

I would like to thank my committee members, Dr. Preetam Ghosh, Dr. Carl Elks, and Dr. Wei Cheng, for serving on my advisory committee and for their insightful comments. I would also like to extend my thanks to other faculty and stuff members of the department of Electrical and Computer Engineering, in particularly to the administrative assistants Stacy E. Metz and Ellen Gresham. Meanwhile, I thank my STAR Lab mates, Tao Lu, Kun Tang, Ping Huang, Pradeep Subedi, and other lab mates Qianbin Xia, Dongwei Wang, Liang Xu. They helped me a lot and we spent a good time together. I thank my friends, Bob and Elaine Metcalf, Jim and Jan Fiorelli, Dick and Janet Andrews, Lex and Kate Strickland, Bud Whitehouse, Ed and Suping Boudreau, Geoffrey and Eunice Chan, for our friendships at Richmond.

I cannot express my gratitude to my wife, Qiong. Without her unfailing support and love, it is impossible to complete this dissertation. I thank my little son Leo for bring joy to the family. His joy is precious to me. I thank my parents, parents in law and my brother for their love, encouraging and support.

TABLE OF CONTENTS

Chapter Page

Acknowledgements ii

Table of Contents iii

List of Tables v

List of Figures vi

Abstract x

1 Introduction 1

1.1 Background and Problem Statement 1
1.2 Proposed Approaches 3

2 Alleviating DRAM Refresh Overhead 6

2.1 Introduction 6
2.2 Background and Motivation 8
2.2.1 DRAM Refresh 9
2.2.2 Refresh Penalty 11
2.2.3 Limitations of Existing Solutions 12
2.3 Architecture and Design 14
2.3.1 DRAM Refresh Characterization 14
2.3.2 The RPC Architecture 15
2.3.3 Cache Design 17
2.3.4 Implementation Overhead 20
2.4 Evaluation 22
2.4.1 Evaluation Methodology 22
2.4.2 Design Space Exploration 23
2.4.3 Single-Core Simulation Results 24
2.4.4 Four-Core Simulation Results 26
2.5 Related Work 27

3 Architecting Die-stacked DRAM as A Part of Memory 31

3.1 Introduction 31

3.2 Background and Motivation 33
3.2.1 OS-managed PoM 34
3.2.2 Hardware-managed PoM 35
3.2.2.1 Line-based PoM 35
3.2.2.2 Page-based PoM 36
3.3 Architecture and Design 37
3.3.1 Remapping Table Design 37
3.3.2 Page Swapping 41
3.3.3 Page Footprints 42
3.3.4 Line Location Prediction 43
3.3.5 Put Everything Together 44
3.3.6 Overhead Comparison 45
3.4 Evaluation 46
3.4.1 Evaluation Methodology 46
3.4.2 Performance Results 48
3.4.3 Hit Ratio and Off-chip Traffic 49
3.4.4 Prediction Accuracy 51
3.4.5 Energy Analysis 52
3.4.6 Sensitivity to RPT Cache Size 54
3.4.7 Sensitivity to Swap Threshold 55
3.5 Related Work 56

4 Page Placement in Die-stacked DRAM/NVM Memory Systems 58

4.1 Introduction 58
4.2 Background and Motivation 60
4.2.1 Emerging Memory Technologies 60
4.2.2 Hybrid Memory Systems 62
4.3 Architecture and Design 65
4.3.1 HBM Cache Design 67
4.3.2 HBM Cache Bypassing 67
4.3.3 Page Migration 69
4.3.4 Write Reduction 71
4.4 Evaluation 72
4.4.1 Evaluation Methodology 72
4.4.2 Performance Results 74
4.4.3 Write Traffic on PCM 76
4.4.4 Energy Consumption 78
4.4.5 Overhead Analysis 79

4.4.6 Sensitivity Study 80
4.4.6.1 Migration Threshold Sensitivity Analysis 80
4.4.6.2 HBM Cache Size Sensitivity Analysis 80
4.5 Related Work 82

5 Conclusions and Future Work 84

5.1 Conclusions 84
5.2 Future Work 85

References 87

Vita 99

LIST OF TABLES

Table Page

1 Refresh related parameters under different DRAM densities. 10

2 tREFI and tRFC parameters in different refresh modes with various DRAM densities. 13

3 Notations used in Equation 2.1. 19

4 Configuration of Simulators. 22

5 Storage Overhead Comparison. 46

6 System Configuration. 47

7 The breakdown of prediction accuracy. 52

8 Workload Statistics. 68

9 Page Eviction Cases in HBM Cache. 72

10 System Parameters. 73

11 Storage Overhead Comparison 80

LIST OF FIGURES

Figure Page

1 The architecture of modern memory systems. The proposed approaches address problems in different memory technologies, which are circled with dotted lines. 4

2 Performance degradation with various chip capacities in the normal temperature range ($\leq 85\,^{\circ}$C). The geometric mean values are 21.7% and 24.7% for the PARSEC and SPLASH-2 benchmark suites, respectively. 11

3 Staggered refresh in ranks within the same channel. 15

4 The RPC architecture. The to-be-read data is cached in an adjacent rank before the target rank is locked and each rank reserves or over-provisions a cache region to store the cached data. All cache regions are organized in a rotated fashion. 16

5 Access locality of PARSEC benchmark suite running on two-rank DRAM system. X-axis represents access sequence and Y-axis is the row number calculated by Equation 2.1. 100 consecutive memory accesses are randomly chosen to show. 18

6 The performance of *RPC* with various cache size, normalized to the 1x mode refresh scheme. 24

7 The performance of *RPC* with various tCI, normalized to the 1x mode refresh scheme. 24

8 Performance comparisons among *FGR*, *RPC* and *No Refresh* schemes with various number of ranks in a single-core system. *RPC* outperforms all *FGR* modes and improves system performance by 8.1% (8.7%) and 9.6% (10.8%) on average for PARSEC and SPLASH-2 benchmark suites in the two-rank (four-rank) system, respectively. 25

9 Performance comparisons among *FGR*, *RPC* and *No Refresh* schemes with various number of ranks in a four-core system. *RPC* outperforms all *FGR* modes and improves system performance by 10.7% (8.6%) and 9.3% (12.2%) on average for PARSEC and SPLASH-2 benchmark suites in the two-rank (four-rank) system, respectively. 27

10 The performance of a state-of-the-art page-based PoM design [47] with different page sizes. All requests are serviced from on-chip memory in the ideal case. On average, the page-based design performs best at the page size of 4KB. 36

11 Direct remapping and corresponding remapping tables. 38

12 The data layout of on-chip memory after co-locating each data line with its corresponding RLT entry. 40

13 Overview of SELF architecture. When the competing counter (CC) is larger than the swap threshold, SELF selectively swaps lines in the requested page according to its page footprint. Otherwise, SELF uses page location to predict the requested line location to reduce latency of off-chip accesses. 44

14 Performance comparisons. On average, CAMEO and PoM improve performance by 9.5% and 9.9%, respectively, while SELF improves performance by 26.9%, which is 85% of the ideal case. 48

15 Two import performance metrics (a) hit ratio of on-chip memory and (b) off-chip traffic. All results are normalized to the baseline. SELF achieves an average hit ratio of 76% while reducing off-chip traffic to 46% of the baseline system. Although PoM obtains the highest hit ratio, 89% on average, it also causes the highest off-chip traffic, 153% on average. 50

16 Energy consumption. On average, CAMEO, PoM and SELF reduce energy per access by 31.3%, 27.6% and 47.9%, respectively. 53

17 Hit ratio of RPT cache across different cache size. 54

18 Performance sensitivity to various swap thresholds (6, 8, 12, and 16). . . . 55

19 Typical hybrid memory system architectures. 63

20 The architecture of DR-HBM. The miss table (MT) and hot page buffer (HPB) are reserved in HBM cache. 66

21 Performance comparisons. On average, CAMEO and RaPP improve performance by 25% and 37.1%, respectively, as compared to the baseline, while DR-HBM improves performance by 63%. 74

22 Hit ratio of fast memory. Baseline, CAMEO, RaPP and DR-HBM achieve an average hit ratio of 4.2%, 56.6%, 63%, and 97.2%, respectively. 75

23 The write traffic on PCM. Compared to the baseline, CAMEO, RaPP and DR-HBM reduce the write traffic on PCM by 29.4%, 26.6% and 89.6% on average, respectively. 77

24 Energy consumption. Compared to the baseline, CAMEO, RaPP and DR-HBM reduce the energy consumption by 6.3%, 3% and 32.9% on average, respectively. 78

25 Performance sensitivity to the migration threshold (32, 64, and 128). . . . 81

26 Performance sensitivity to the HBM cache size (64MB, 128MB, and 256MB). 81

Abstract

IMPROVING THE PERFORMANCE AND ENERGY EFFICIENCY OF EMERGING MEMORY SYSTEMS

By Yuhua Guo

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at Virginia Commonwealth University.

Virginia Commonwealth University, 2018.

Advisor: Weijun Xiao, Ph.D.

Assistant Professor, Virginia Commonwealth University

Co-advisor: Xubin He, Ph.D.

Professor, Temple University

Modern main memory is primarily built using dynamic random access memory (DRAM) chips. As DRAM chip scales to higher density, there are mainly three problems that impede DRAM scalability and performance improvement. First, DRAM refresh overhead grows from negligible to severe, which limits DRAM scalability and causes performance degradation. Second, although memory capacity has increased dramatically in past decade, memory bandwidth has not kept pace with CPU performance scaling, which has led to the memory wall problem. Third, DRAM dissipates considerable power and has been reported to account for as much as 40% of the total system energy and this problem exacerbates as DRAM scales up.

To address these problems, 1) we propose *Rank-level Piggyback Caching (RPC)* to alleviate DRAM refresh overhead by servicing memory requests and refresh operations in parallel; 2) we propose a high performance and bandwidth efficient approach, called

SELF, to breaking the memory bandwidth wall by exploiting die-stacked DRAM as a part of memory; 3) we propose a cost-effective and energy-efficient architecture for hybrid memory systems composed of high bandwidth memory (HBM) and phase change memory (PCM), called Dual Role HBM (DR-HBM). In DR-HBM, hot pages are tracked at a cost-effective way and migrated to the HBM to improve performance, while cold pages are stored at the PCM to save energy.

CHAPTER 1

INTRODUCTION

1.1 Background and Problem Statement

The capacity of main memory keeps increasing, which is mainly driven by the growing memory requirements of new applications, and the increasing number of processing cores in a single chip. Dynamic Random Access Memory (DRAM) has been used as the main memory in computer systems for decades. However, DRAM-based memory systems are mainly facing three scalability problems.

First, DRAM cells leak charge over time, causing stored data to be lost. Therefore, periodic refreshes are required to ensure data integrity. Commodity DRAM refreshes cells at rank level, resulting in an entire rank being unavailable during a refresh period. As DRAM density keeps increasing, more rows need to be refreshed during a single refresh operation, which causes higher refresh latency and significantly degrades the overall memory system performance [1]. Currently refresh overhead has become the biggest restriction for DRAM scalability, making it increasingly important to reduce refresh overhead [2].

Second, although memory capacity has increased dramatically in past decade, memory bandwidth has not kept pace with CPU performance scaling, which has led to the memory wall problem [3]. Die-stacked DRAM (a.k.a., on-chip DRAM) provides much higher bandwidth and lower latency than off-chip DRAM. It is a promising technology to break the "memory wall". However, on-chip DRAM is not large enough to fully replace off-chip DRAM. Therefore, on-chip DRAM is used either as a cache (i.e., DRAM cache) or as a part of memory (PoM). A DRAM cache design would suffer from more page faults than a PoM design as the DRAM cache cannot contribute towards capacity of main memory. In the meanwhile, obtaining high performance requires PoM systems to swap requested data to the on-chip DRAM. Existing PoM designs fall into two categories — line-based and page-based. The former ensures low off-chip bandwidth utilization but suffers from a low hit ratio of on-chip memory due to limited temporal locality. In contrast, page-based designs achieve a high hit ratio of on-chip memory while at the cost of moving large amounts of data between on-chip and off-chip memories, leading to increased off-chip bandwidth utilization and significant system performance degradation. How to achieve a similar high hit ratio of on-chip memory as page-based designs and eliminate excessive off-chip traffic involved is a big challenge.

Last, traditional DRAM-based memory systems are also facing the power issue besides the memory wall problem. DRAM dissipates considerable power and has been reported to account for as much as 40% of the total system energy [4, 5, 6]. This problem exacerbates as DRAM capacity increases. Therefore, emerging non-volatile memories (NVMs), such as Spin-Transfer Torque RAM (STT-RAM) and Phase Change Memory (PCM), are gaining interest as DRAM alternatives as they have near-zero standby power. However, NVMs have lower memory bandwidth and longer access latency than DRAM, aggravating the memory wall problem. Since die-stacked DRAM has the potential to break the memory wall but its capacity is insufficient to fully replace DRAM memory, a die-stacked DRAM/NVM hybrid memory system could be a promising way to build a high performance, large capacity, and energy efficient memory system. In order to fully exploit high performance (i.e. high bandwidth and low latency) offered by die-stacked DRAM and large capacity offered by NVM, hot pages should be migrated to die-stacked DRAM to improve performance and cold pages should be stored in NVM to save energy. Therefore, how to identify hot pages is very critical. Prior work regarding DRAM/NVM hybrid memory systems [7, 8, 9, 10] has been proposed. However, the ways used to track page hotness in prior work are costly due to redesigning the memory controller (MC) or extending translation lookaside buffer (TLB). Moreover, the hot page stays at NVM until its access count exceeds the migration threshold, which could miss lots of opportunities to improve performance. Therefore, it is very promising to design a cost-effective and energy efficient architecture for die-stacked DRAM/NVM memory systems.

1.2 Proposed Approaches

Figure 1 shows the architecture of modern memory systems, which could consist of traditional DRAM, die-stacked DRAM and emerging NVMs. As stated in Section 1.1, each component has its own limitations. In order to build a high performance, large capacity, and energy efficient memory system, we propose three approaches to address these three scalability problems, respectively, as shown in Figure 1.

First, to mitigate DRAM refresh overhead, we propose a caching scheme, called *Rank-level Piggyback Caching*, or *RPC* for short, based on the fact that ranks in the same channel are refreshed in a staggered manner. The key idea is to cache the to-be-read data in a rank (e.g. Rank 1) to its adjacent rank (e.g. Rank 2) before Rank 1 is locked for refresh. Each rank reserves or over-provisions a very small area, denoted as a cache region, to store the cached data. The cache regions from all ranks are organized in a rotated fashion. In other words, the cached data for the last rank is stored in the first rank. When a read request arrives at a rank undergoing refresh, the memory controller first checks the cache region in the next rank in the same channel; if the requested data is cached, the memory controller services the request from the cache without waiting for the refresh operation to complete, which reduces memory access latency and improves system performance.

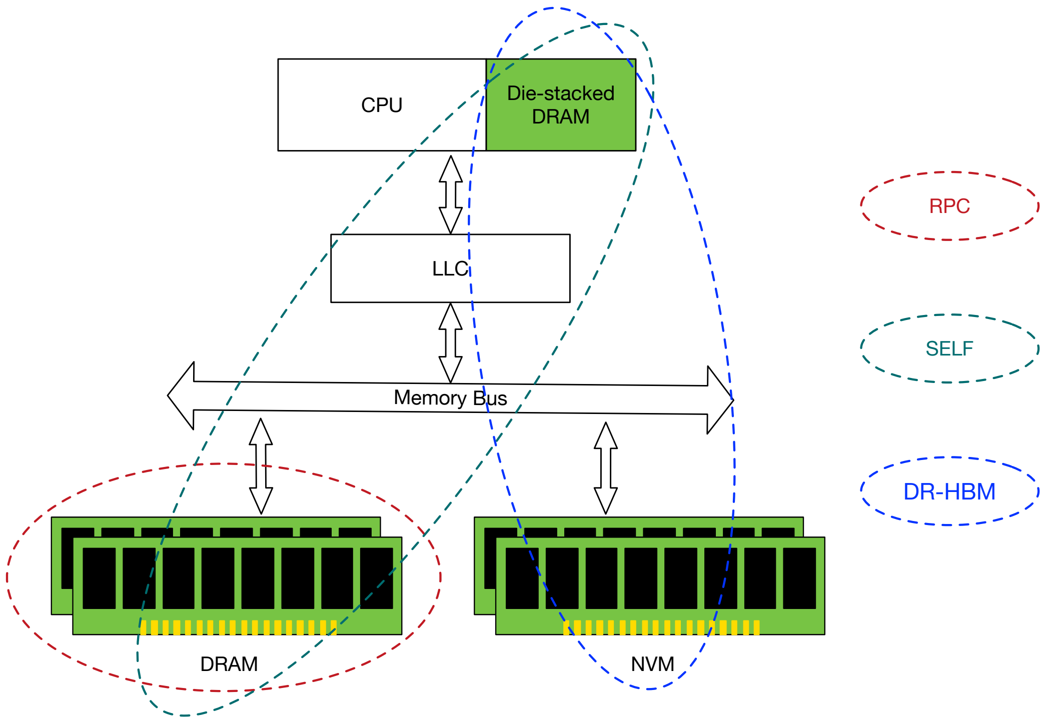


Fig. 1. The architecture of modern memory systems. The proposed approaches address problems in different memory technologies, which are circled with dotted lines.

Second, to achieve a similar high hit ratio of on-chip memory as page-based designs, and eliminate excessive off-chip traffic involved, we propose SELF, a high performance and bandwidth efficient approach. The key idea is to SElectively swap Lines in a requested page that are likely to be accessed according to page Footprint, instead of blindly swapping an entire page. In doing so, SELF allows incoming requests to be serviced from the on-chip memory as much as possible, while avoiding swapping unused lines to reduce memory bandwidth consumption.

Last, we propose a cost-effective and energy-efficient architecture for die-stacked

DRAM/NVM memory systems, especially for HBM/PCM memory systems, called Dual Role HBM (DR-HBM). In DR-HBM, the HBM plays two roles and is divided into two parts. A small portion of which, called HBM cache, is used as a cache for the PCM. The remaining HBM is used as a part of main memory. Furthermore, the HBM cache is also used to track page hotness without additional hardware support. In order to improve performance and reduce writes to the PCM, we propose three techniques. First, CSM (cache on the second miss) increases the effectiveness of HBM cache and reduces PCM traffic by avoiding to cache singleton pages which contain only single useful data blocks; Second, hot pages are migrated in batches to amortize TLB shoot-down overhead; Third, we propose Hot First LRU (HF-LRU) page replacement policy and increase the weight of write operations to reduce writes to the PCM.

CHAPTER 2

ALLEVIATING DRAM REFRESH OVERHEAD

2.1 Introduction

Modern main memory is primarily built using dynamic random access memory (DRAM) cells. A DRAM cell consists of one access transistor and one capacitor. Each DRAM cell stores one bit of data as electrical charge in the capacitor; over time charge will leak from the capacitor and can cause data loss. Therefore, DRAM requires an operation called *refresh* that periodically restores electrical charge in capacitors to ensure data integrity.

Each DRAM cell is refreshed every 64ms (or 32ms above 85 °C) as specified by the JEDEC standards [11]. This time period is called retention time. All DRAM rows are refreshed within this time period. The total time spent on refresh operations is proportional to the number of rows in a DRAM device (a.k.a. a chip). Initially all DRAM rows are refreshed sequentially within one refresh operation, which causes long periods of memory unavailability. To avoid this long latency, all DRAM rows in a bank are divided into 8K groups, and each group is refreshed within a time period of 64ms/8K = 7.8μsec (3.9μsec at high temperatures). This time duration is called *refresh interval*, denoted as *tREFI*. The memory controller sends a refresh command to DRAM devices once every *tREFI*. The time duration of one refresh command is referred to as *refresh cycle*, denoted as *tRFC*. Each DRAM row is composed of thousands of bits. The size of each row is referred to as page size, and the capacity of a DRAM device is the number of rows in a device times the page size. The page size has remained between 1KB to 2KB for several DRAM generations while the number of rows per device has scaled linearly with DRAM device capacity. As the capacity of the DRAM device increases and the refresh interval remains unchanged, the refresh cycle increases quickly since more DRAM rows need to be refreshed in each refresh operation. As a result, it takes a longer time and more energy to complete a refresh.

The observation that motivates this work is that commodity DRAM refreshes cells at rank level. As a result, an entire rank is locked up and cannot serve any pending memory request while being refreshed. Thus, a read request arriving at a rank that is being refreshed is forced to wait until the refresh operation is completed, which would increase read latency and degrades system performance. As memory technology scales to higher densities, for example, 16Gb DRAM device that has been defined in DDR4 SDRAM standard [12], performance degradation and energy consumption that attributes to refresh grow from negligible to severe. Currently refresh overhead has become the biggest restriction for DRAM scalability, making it increasingly important to reduce refresh overhead [2].

Our goal is to alleviate the performance degradation due to DRAM refresh operations. As ranks within a channel are refreshed in a staggered fashion, while one rank is being refreshed, the remaining ranks can still service memory requests. In light of this, we propose *RPC (Rank-level Piggyback Caching)* [13], a scheme that allows concurrent refresh and memory access in a DRAM system. The basic idea is to cache the data that will be likely accessed during the next refresh period to an adjacent rank in the same channel before the target rank is locked. As such, read requests issued to a rank which is being refreshed can be served from another rank without waiting for the refresh to complete. Without being blocked during refresh, the read latency can be significantly shortened and system performance in terms of instructions per cycle can be improved.

2.2 Background and Motivation

A modern DRAM-based memory system has one or more memory controllers, each of which manages one or more channels. Each channel has independent command, address, and data buses, allowing different channels to be accessed concurrently. Each channel is composed of a number of ranks that share the channel bandwidth and operate simultaneously to service memory requests. Each rank can further divided into DRAM chips and within each chip, there are a number of banks (typically 8 banks) which can be accessed in parallel as well. Each bank consists of a two-dimensional array of DRAM cells. A DRAM cell consists of one access transistor and one capacitor. Each DRAM cell can store one bit of data as electrical charge in the capacitor. All the access transistors in the same column connect their capacitors to a wire called *bitline*. An access transistor is controlled by a wire called *wordline* which is shared by a row of DRAM cells. DRAM cells sharing a wordline form a DRAM row. Each bitline connects to a sense amplifier, a row of which is known as *row buffer* which is used to sense and amplify the voltage of each bitline [14]. A bank can be further sub-divided into many *subarrays*, with each having its own row buffer (a.k.a. local row buffer). However, only one subarray can be accessed at a time since all subarrays share the global bitlines. All subarrays' row buffers are connected to a global I/O buffer. The memory controller reads/writes data from/to the I/O buffer through the bank's I/O bus.

A bank supports four types of operations: activation (ACT or RAS), read/write (CAS), precharge (PRE) and refresh (REF). To retrieve data from DRAM cells, the row which contains the requested data must first be activated (or opened) to put the entire row's content into the row buffer through bitlines. Then the requested data can be retrieved from the row buffer by decoding the column address. If subsequent requests access the same row in the same bank, a row hit happens, otherwise a row miss occurs. There are two row buffer management policies: open page and close page [15]. In the open page mode, the accessed row is not closed until a different row within the same bank is accessed. The activation operation can be obviated if it is a row hit. Otherwise the memory controller first needs to close the row and precharges bitlines for the next activation, which causes extra latency and power consumption [16]. Therefore, the open page mode is more suitable for workloads with good access locality. In contrast, in the close page mode, the memory controller proactively closes the row and precharges bitlines as soon as an access is over, which provides a consistently fair latency. It is clear that the close page mode is beneficial for workloads with poor access locality, such as, the multi-core environments where there could be mutual interferences from different applications.

2.2.1 DRAM Refresh

Due to capacitor current leakage, each DRAM cell needs to be refreshed periodically to maintain data integrity. DRAM cells are refreshed in the unit of a row. A refresh operation consists of two steps. First, data in the refreshing row is read out and written back to the cells to restore each capacitor's charge, which is actually the activation operation. Then the bitlines are precharged for the next refresh operation. Hence, the refresh operation is functionally equivalent to an activation plus a precharge operation. The time taken to refresh one row is known as *row cycle time (tRC)*, which is the time used to activate and precharge one row. DRAM cells have varying retention time [17] and the JEDEC standards specify a minimum of 64ms (or 32ms at high temperatures), which means all DRAM rows must be refreshed within this time period. Initially, there were relatively few rows in a bank, therefore it was viable to refresh all DRAM rows in succession within retention time. This is referred to as *bursty refresh mode*. However, refreshing all DRAM rows in bulk incurs high latency, which becomes unacceptable as the number of DRAM rows in a bank increases to tens of thousands. To avoid this high latency, JEDEC supports *distributed refresh mode*, in which DRAM rows in a bank are divided into 8K groups. The memory controller issues a refresh command to refresh one group every *tREFI* which is equal to 64ms/8k = 7.8μsec (3.9μsec at high temperatures). The time spent on refreshing one group is known as *refresh cycle* or *tRFC*, which is a function of *tRC*. The *tREFI* has remained unchanged for several generations, but *tRFC* increases linearly as DRAM chip scales to higher densities. *tRFC/tREFI* is defined as *refresh duty cycle* (RDC) [18], the percentage of time that the DRAM system spends on doing refresh. Table 1 shows several refresh related parameters under different chip densities. Values for 16Gb chip are extrapolated. The number of rows (Row Num.) in a bank doubles as the density of DRAM chip is doubled, as does the number of rows refreshed in each refresh operation (Rows/REF) because the total number of refresh operations (Refresh Num.) in the retention time period remains constant. Therefore, the tRFC increases linearly as DRAM density increases, as does the RDC.

Table 1. Refresh related parameters under different DRAM densities.

Chip Density	1Gb [19]	2Gb [20]	4Gb [21]	8Gb [22]	16Gb
Retention Time(ms)	64/32	64/32	64/32	64/32	64/32
Row Num.	16K	32K	64K	128K	256K
Refresh Num.	8K	8K	8K	8K	8K
Rows/REF	2	4	8	16	32
tREFI(μs)	7.8/3.9	7.8/3.9	7.8/3.9	7.8/3.9	7.8/3.9
tRC(ns)	51	51	51	51	51
tRFC(ns)	110	160	260	350	450
tRFC/tREFI	1.41%	2.05%	3.34%	4.34%	5.77%
tRFC/tREFI ($> 85\,^\circ$C)	2.82%	4.11%	6.68%	8.68%	11.54%

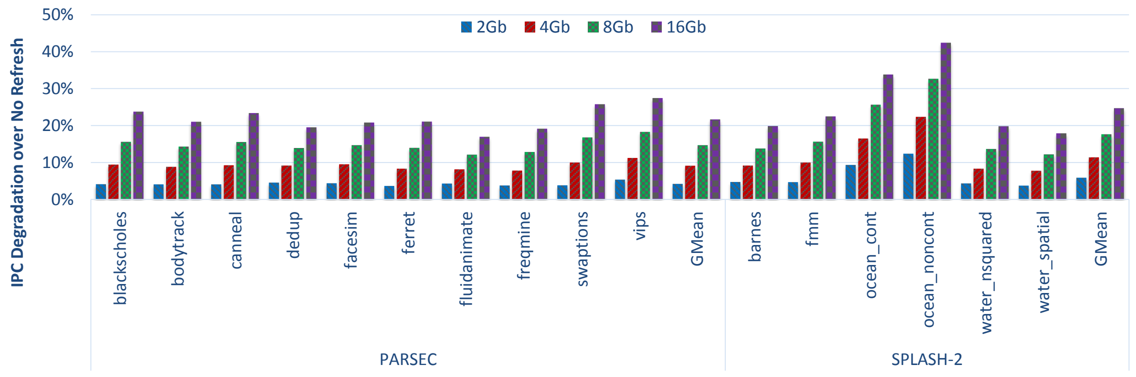


Fig. 2. Performance degradation with various chip capacities in the normal temperature range ($\leq 85\,^{\circ}$C). The geometric mean values are 21.7% and 24.7% for the PARSEC and SPLASH-2 benchmark suites, respectively.

2.2.2 Refresh Penalty

In theory, a refresh scheme at any granularity is valid as long as all cells can be refreshed timely. However, commodity DRAM refreshes cells at rank level, which means all chips in a rank and all banks in a chip are refreshed in a lockstep manner. A rank which is undergoing a refresh cannot serve any memory access. In other words, refresh and memory access are mutually exclusive to each other at the granularity of a rank, which is the main contributing factor of refresh penalty [23].

As shown in Table 1, the *tRFC* is growing dramatically with chip density increases, which aggravates the refresh penalty. The RDC increases to 11.54% when DRAM chip capacity increases to 16Gb, which means a rank spends 11.54% of the time refreshing. Therefore, the refresh overhead is no longer trivial and will degrade system performance significantly. Figure 2 shows the performance degradation due to refresh compared to an ideal case without refresh. When DRAM chips scale to 16Gb, the performance degradation can be as high as 42.4% for memory latency sensitive workloads and the geometric mean values are 21.7% and 24.7% for the PARSEC and

SPLASH-2 benchmark suites, respectively. The performance degradation becomes more severe as DRAM chip density increases, demonstrating that it is increasingly important to alleviate refresh overhead in high density memory.

2.2.3 Limitations of Existing Solutions

JEDEC specifications define the refresh scheduling flexibility: up to eight refresh commands can be postponed or issued in advance. Stuecheli et al. [1] proposed *Elastic Refresh (ER)*, which leverages the 8-tREFI refresh scheduling flexibility to hide refresh penalty. *ER* prioritizes DRAM accesses over DRAM refresh operations by postponing refresh operations to decrease the probability of conflicts between them. Every refresh command needs to wait an elastic window period determined by the average idle time of the rank and the number of already postponed refresh commands to avoid interfering with demand requests. In fact, previous work DUE [24] makes use of the flexibility in issuing refresh operations by scheduling them when the rank queues are idle. *ER* further defers refresh operations for an extra time period after the rank becomes idle to service the incoming requests with priority, but the postponed refresh commands need to be enforced immediately when the number of postponed refresh commands hits the 9-tREFI limitation. However, *ER* becomes less effective as DRAM scales [23], [25]. The increasing *tRFC* makes the refresh latency hard to be hidden as the average rank idle period is shorter than *tRFC*. Moreover, *ER* incurs extra delay when it incorrectly predicts a time period as idle when it actually has pending requests.

A read request arriving at a rank which is undergoing a refresh operation needs to wait until the refresh operation completes. In the worst case scenario it must wait for *tRFC*. To shorten the *tRFC*, *Fine Granularity Refresh (FGR)* was proposed in the DDR4 SDRAM Standard [12]. *FGR* defines three refresh modes with different refresh rates (i.e. tREFI). They are 1x mode, 2x mode and 4x mode respectively, and the memory controller can switch between them on the fly. The 1x mode is the same as the traditional refresh scheme defined in [11], in which each refresh command is issued every $\text{tREFI}_{1x} = 7.8\mu\text{sec}$. The 2x and 4x modes require that refresh commands are issued two and four times as frequently as 1x mode. Due to the increasing number of refresh commands, fewer rows need to be refreshed during a single refresh. Therefore, the *tRFC* is reduced accordingly in the 2x and 4x modes. Table 2 shows that the *tREFI* and *tREC* parameters under different refresh modes vary with different DRAM densities. Values for 16Gb DRAM chips which are not decided in the DDR4 standard [12], are extrapolated based on the previous values of low densities; we use the 8Gb chip as the default DRAM chip in our experiments. The refresh cycle is reduced as the refresh rate increases. We implement *FGR* in DRAMSim2 [26] and run different benchmarks in PARSEC 2.1 [27] and SPLASH-2 [28] benchmark suites to observe the performance of *FGR*.

Table 2. tREFI and tRFC parameters in different refresh modes with various DRAM densities.

Refresh Mode	Parameter	2Gb	4Gb	8Gb	16Gb
1x mode	tREFI_{1x} (μs)	7.8/3.9	7.8/3.9	7.8/3.9	7.8/3.9
	tRFC_{1x} (ns)	160	260	350	450
2x mode	tREFI_{2x} (μs)	3.9/1.95	3.9/1.95	3.9/1.95	3.9/1.95
	tRFC_{2x} (ns)	110	160	260	350
4x mode	tREFI_{4x} (μs)	1.95/0.975	1.95/0.975	1.95/0.975	1.95/0.975
	tRFC_{4x} (ns)	90	110	160	260

Figure 8 shows the performance (IPC) of *FGR* in the two-rank and four-rank systems. All results are normalized to the 1x mode. In the two-rank DRAM system, the performance of both 2x and 4x modes are worse than 1x mode. There are two reasons: 1) *tRFC* reduces linearly but not proportionally as the refresh rate increases, which means 2xtRFC$_{2x}$ is larger than 1xtRFC$_{1x}$, and 2xtRFC$_{4x}$ is larger than 1xtRFC$_{2x}$ too; 2) Both PARSEC and SPLASH-2 benchmark suites are composed of multi-thread and memory-sharing programs, which are memory-intensive workloads (detailed analysis can be found in Section 2.4.3). In the memory-intensive case, the average rank idle period is even shorter than tRFC$_{4x}$, so the refresh penalty cannot be hidden. Thus an application only needs to wait for one tRFC$_{1x}$ in the 1x mode, it may experience 2 (or 4) stalls in the 2x (or 4x) mode because of the increasing refresh rate. The total time spent on doing refresh is longer than that in 1x mode, which causes performance degradation in the 2x mode and 4x mode. In the four-rank system, the average performance of 2x and 4x modes are better than that in the two-rank system, respectively. For *dedup* benchmark, both 2x and 4x modes even perform better than 1x mode. The reason is that the increasing number of ranks reduces the probability of conflicts between memory requests and DRAM refreshes, thereby the DRAM system can get benefit from the reduced *tRFC*. Overall, *FGR* is not suitable for memory-intensive workloads.

All refresh schemes discussed above could become ineffective when DRAM scales to high densities or workloads become memory-intensive. Therefore, we propose *RPC* to mitigate refresh overhead.

2.3 Architecture and Design

2.3.1 DRAM Refresh Characterization

A DRAM-based memory system usually consists of multiple ranks. Each rank works independently even though all of the ranks share the same channel bandwidth. Commodity DRAM refreshes at rank level. The memory controller issues a refresh command to a rank every *tREFI*. Each refresh operation lasts for a duration of *tRFC*.

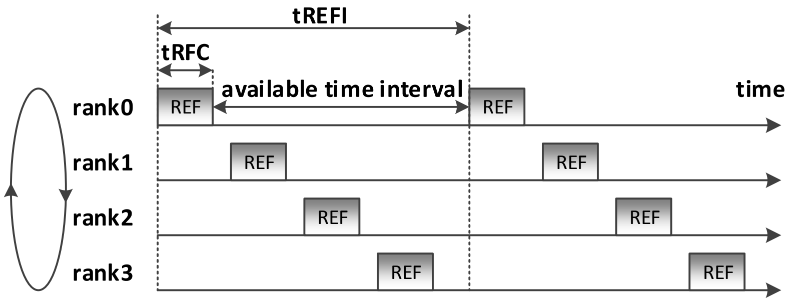


Fig. 3. Staggered refresh in ranks within the same channel.

Thus the available time interval between two contiguous refresh operations equals to $tREFI - tRFC$. When a rank receives a refresh command from memory controller, all banks in the rank are refreshed concurrently. Therefore, no memory access is allowed to a rank where a refresh is undergoing. Fortunately, only one rank is allowed to be refreshed at a time to meet power budget. All ranks in the same channel are refreshed in a staggered fashion, as shown in Figure 3. There is no time overlap between different refresh commands occurred to different ranks. In this case, other ranks in the same channel can still service DRAM accesses normally when one rank is being refreshed. However, a request arriving at the rank which is being refreshed still needs to wait until the refresh operation completes, which increases the read latency. In the worst case, the waiting time is *tRFC*, which is an order of magnitude longer than a typical read response. Hence, system performance is degraded significantly and the longer running time in turn increases static energy consumption. It is increasingly important to mitigate refresh overhead.

2.3.2 The RPC Architecture

The design goal of *RPC* is to alleviate refresh overhead by serving memory requests and refresh operations that are issued to the same rank in parallel. Since ranks in the same channel are refreshed in a staggered manner, the remaining ranks

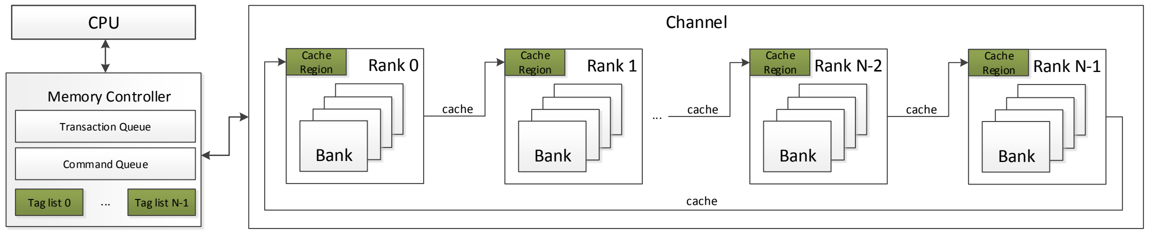
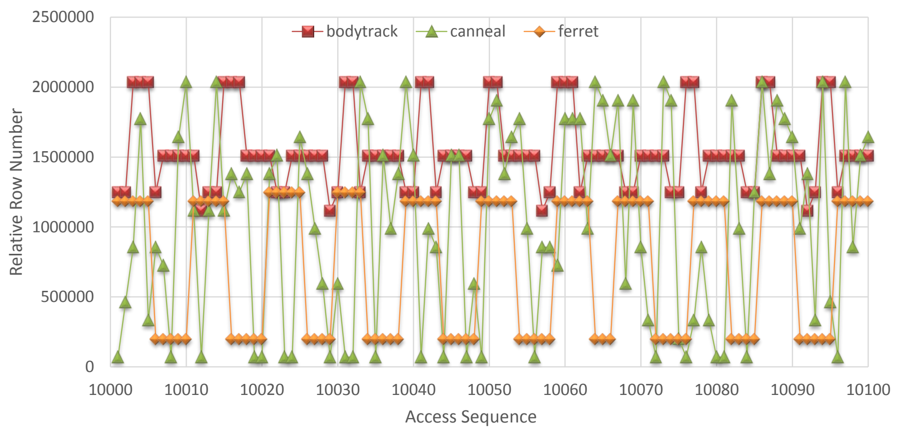


Fig. 4. The RPC architecture. The to-be-read data is cached in an adjacent rank before the target rank is locked and each rank reserves or over-provisions a cache region to store the cached data. All cache regions are organized in a rotated fashion.

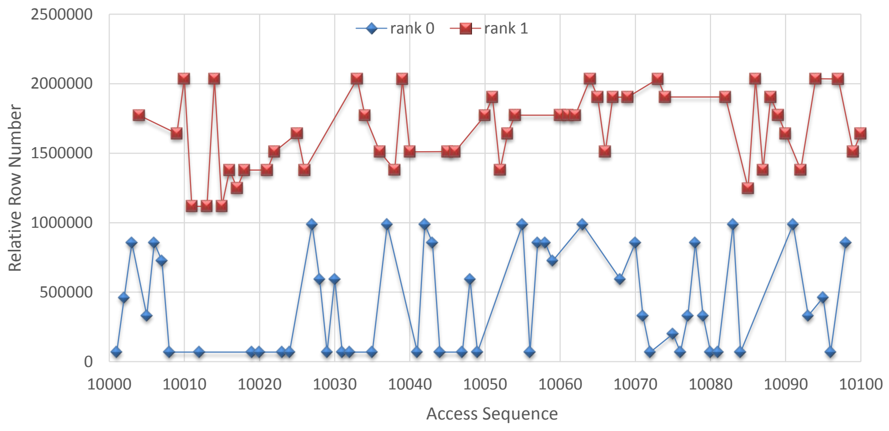
in the same channel are still available when a rank is being refreshed. Based on this DRAM characteristic, we propose *RPC* to mitigate refresh overhead. The idea is to cache those data which will be read during the next refresh period, and store it to an adjacent rank before the target rank is locked. The data is populated to the cache in a piggyback manner in which data is cached when it is accessed, rather than prefetching it to the cache in the adjacent rank in a bursty fashion. In our design, only read requests are taken into consideration and this is due to two reasons: 1) read requests are latency sensitive because applications cannot proceed until data is retrieved; 2) write requests can be cached in a write buffer and flushed to memory in batches asynchronously. The design of *RPC* is shown in Figure 4. Each rank reserves or over-provisions an area, called a cache region, to store the cached data; the size of the cache region is a configurable parameter in later performance evaluations so that we can gauge the impact of cache size. The cache region placed on rank i only services memory requests addressed to its previous rank i - 1, and all cache regions are organized in a rotated fashion. In particular, cached data from the last rank, N - 1, is stored at rank 0. When a read request arrives at a rank where a refresh is undergoing, the memory controller first checks the cache region in the “next” rank in the same channel; if the required data is cached, the memory controller can serve the read request without waiting for the refresh operation to complete, which reduces refresh overhead and improves system performance. However, the effectiveness of *RPC* depends on cache hit ratio.

2.3.3 Cache Design

As shown in Figure 4, the CPU sends requests to the memory controller in the form of transactions. First, a transaction is enqueued into the transaction queue if there is spare space; then it is translated into DRAM commands (e.g. ACT, CAS, PRE etc.) and enqueued into the command queue. To achieve high row hit ratio, modern memory controllers commonly adopts FR-FCFS (first-ready first-come-first-serve) scheduling policy [29], which prioritizes DRAM commands that cause row hits over other commands, including those that were issued earlier. If no command results in a row hit, then FR-FCFS schedules commands according to arrival sequence, i.e., FCFS. A key advantage of this scheduling policy is that it retains the temporal locality in the application access pattern. Meanwhile, the row hit ratio is also related to the address mapping policy. In this work, we use the "channel:row:column:bank:rank" address mapping policy, which is commonly deployed in state-of-the-art memory controllers. In this address mapping, the row bits are placed as MSBs to maximize the row hit ratio while keeping ranks and banks interleaving. We run the full PARSEC benchmark suite to observe the memory access locality. For brevity, three representative benchmarks are chosen to be shown in Figure 5a, and other benchmarks have similar results as the *ferret* benchmark. The Y axis represents the relative row number in a channel, which can be calculated by Equation 2.1. Notations used in the equation are listed in Table 3.



(a) Access locality of three representative benchmarks.



(b) Access locality of the *cannel* benchmark in each rank.

Fig. 5. Access locality of PARSEC benchmark suite running on two-rank DRAM system. X-axis represents access sequence and Y-axis is the row number calculated by Equation 2.1. 100 consecutive memory accesses are randomly chosen to show.

$$row\_num = NUM\_ROWS * NUM\_BANKS * rank + NUM\_ROWS * bank + row \tag{2.1}$$

All benchmarks show very strong temporal locality except *canneal*, which shows poor access locality. However, the cache region is only used for the to-be-refreshed rank instead of the global scope (i.e. the whole channel). We further break down

Table 3. Notations used in Equation 2.1.

Variables	Description
NUM_ROWS	the total number of row in a bank
NUM_BANKS	the total number of bank in a rank
rank	accessed rank number
bank	accessed bank number
row	accessed row number
row_num	relative row number in a channel

the access pattern of *canneal* benchmark and study the locality of data access on each individual rank. From Figure 5b, we can observe that the temporal locality in each rank is also good even though the overall perceived temporal locality is poor. All of these memory accesses happen after the LLC (last level cache), although some of them have good temporal locality, which means all accesses are missed in the LLC. That is because the size of a row is much larger than a CPU cache line, and a row typically contains 16 to 32 CPU cache lines. Therefore, it is possible that the successive requests access different CPU cache lines but the same DRAM row.

Based on the above observations, DRAM accesses have a strong temporal locality, which means recently accessed rows will likely be accessed again in the near future. Therefore, the cache region is implemented as a LRU (Least Recently Used) cache, where the least recently used data will be evicted and recently accessed data will be saved. We note that the cache region is different from caches in the CPU. CPU caches are used to speed up accesses while the cache region in *RPC* is used to increase data availability during the interval when the target rank is being refreshed, and the entire cache region will be invalidated after the refresh completes. As mentioned in Section 2.2, the granularity of read/write operations in a bank is a DRAM row, thus the size of a cache line is set to the same as the size of a DRAM row (i.e. page size). As shown in Figure 3, the piggyback caching is executed during the available time interval between two continuous refresh operations. Since the cache regions adopt LRU algorithm, in which only recently accessed rows will be cached, we start the piggyback caching a number of cycles ahead of next refresh operation, and end at the beginning of next refresh operation. This time interval is called the caching interval, the length of which is referred to as tCI. The tCI is also configurable. For example, a rank starts to be refreshed at t, the next refresh operation will start at $t+tREFI$. So the piggyback caching begins at $t+tREFI-tCI$, and ends at $t+tREFI$. Algorithm 1 shows the pseudocode of *RPC* in each rank.

2.3.4 Implementation Overhead

To implement *RPC*, each rank needs to have a dedicated cache region or be over-provisioned to accommodate cached data, and the size of it should be an integer multiple of the page size. To track which row is cached, the memory controller also needs to maintain a tag list for each cache region. Each entry is a 64-bit physical address which consists of the channel number, rank number, bank number, row number and column number. The number of entries in each tag list equals to the cache size divided by the page size (i.e. the total number of rows in the cache region). Therefore, the total storage overhead in the memory controller is determined by the cache size, the number of ranks in a channel, and the number of channels. For example, in a single channel and four-rank DRAM system, if the cache size is 16KB and the page size is 1KB, then the total storage overhead in the memory controller is 512 bytes, which is negligible. In addition, copying data to an adjacent rank consumes energy. However, the energy consumption is negligible due to the short caching interval, as demonstrated in Section 2.4.2. *RPC* can reduce the runtime of applications, as a result, it reduces the static energy consumption of DRAM system.

Algorithm 1 The work procedure of *RPC*

```
Input: $tCI$, cache size
Variables: curretn time $t$, next refresh time $tNR$
 1: while DRAM is running do
 2:     while $tNR$ - $tCI \leq t < tNR$ do                    ▷ caching start
 3:         if a memory request comes then
 4:             if the accessed row is cached then
 5:                 move the cache line to list head;
 6:             else
 7:                 if cache region is not full then
 8:                     copy the accessed row to a spare cache line;
 9:                     inset the cache line to list head;
10:                 else
11:                     invalidate the tail cache line;
12:                     copy the accessed row to the tail cache line;
13:                     insert the last cache line to list head;
14:                 end if
15:             end if
16:         end if
17:         $t$++;
18:     end while                                              ▷ caching over
19:     while $tNR \leq t < tNR$ + tRFC do                     ▷ refresh start
20:         if a memory request comes then
21:             if it is a read request then
22:                 if the required row is cached then
23:                     return the cache line to the CPU;
24:                 else
25:                     wait until the refresh is completed;
26:                 end if
27:             else
28:                 handled normally;
29:             end if
30:         end if
31:         $t$++;
32:     end while                                              ▷ refresh over
33:     if $t \geq tNR$ + tRFC then
34:         invalidate the entire cache region;
35:         $tNR = tNR$ + tREFI;
36:     end if
37:     memory requests are handled normally;
38:     $t$++;
39: end while
```

2.4 Evaluation

2.4.1 Evaluation Methodology

To evaluate the effectiveness of our approach, we implement *RPC* in DRAMSim2 [26] and together use MARSSx86 [30] as the front-end processor to run benchmarks. The detailed system configuration is shown in Table 4. The PARSEC 2.1 [27] and SPLASH-2 [28] benchmark suites are used to evaluate our approach. Both of them are multi-thread and memory-sharing benchmark suites. We run all benchmarks for 100 million cycles to warm up the cache and the following 100 million cycles to collect the statistics. The 8Gb DRAM chip is used in our evaluation, and the DRAM chip parameters are set according to the Microns data sheet [22]. The refresh related parameters are listed in Table 1. All simulations are run under normal temperature range ($\leq 85\,^{\circ}\mathrm{C}$). The instructions-per-cycle (IPC) is used as the performance metric throughout the evaluation.

Table 4. Configuration of Simulators.

Processor	1/4 cores, 4GHz, out-of-order 128-entry instruction window
L1-D/L1-I Cache	128KB/128KB, 8-way associative
LLC	64B cache-line, 8-way associative shared, 2MB
Memory Controller	32/32-entry transaction/command queue FR-FCFS [29], open page policy, 64bits I/O bus channel:row:column:bank:rank address mapping
DRAM	DDR3-1333 [22], 8Gb 1 channel, 2/4 ranks per channel 8 banks/rank, 128K rows/bank, 1024 columns/row

For comparisons, we also implement state-of-the-art *FGR* [12] and *No Refresh* refresh schemes in DRAMSim2. The *FGR* includes three different refresh modes with different refresh rates, which are 1x, 2x and 4x modes. Note the 1x mode is the conventional refresh scheme in modern memory controllers. The refresh related parameters of each mode are listed in Table 2. The *No Refresh* scenario is an ideal case where there is no refresh operation. The reason for assessing *No Refresh* is to quantify the best possible performance in terms of IPC. In the runs, we also vary the number of ranks since our design targets rank-level caching and this parameter has a large impact on the overall system performance. As multi-core becomes increasingly prevalent as a means to further increase flops and hence main memory can be shared by multiple cores, contention in the memory system is anticipated to be more severe. Therefore in the evaluations, we also test a four-core scenario to gauge the scalability of our approach.

2.4.2 Design Space Exploration

In our design, both the size of cache region (a.k.a. cache size) and the length of caching interval (tCI) are configurable. As the system performance and overhead are sensitive to these two parameters, we conduct a design space exploration to determine the optimal values for the two parameters before the performance evaluation. The PARSEC 2.1 benchmark suite is used to evaluate. To determine the optimal cache size, memory controller starts caching data at the end of last refresh (i.e. $tCI = tREFI - tRFC$). As shown in Figure 6, with 8KB, 16KB and 32KB cache region, *RPC* improves the system performance by 4.9%, 8.1% and 8.6% on average, respectively. To balance between hardware cost and performance, 16KB is used as the optimal value for cache size, which is equivalent to 16 rows. Based on the optimal cache size, we test three different values of tCI, which are integer multiple of the refresh cycle ($tRFC$). As shown in Figure 7, the 1xtRFC tCI is sufficient to achieve almost all the performance margin due to the usage of LRU cache algorithm and the limited cache size. Therefore, 1xtRFC is chosen as tCI for subsequent runs.

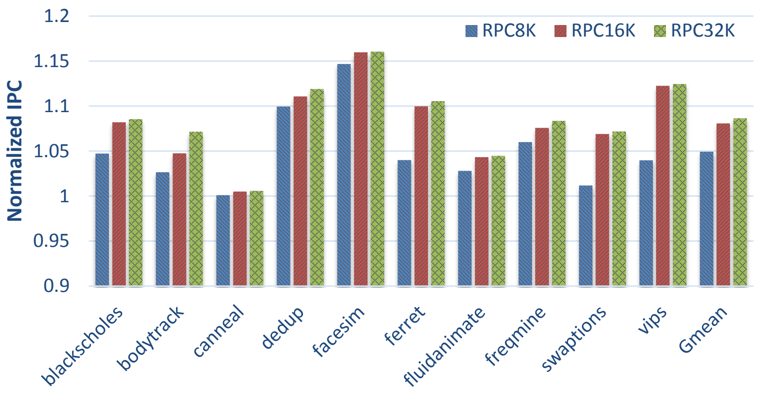


Fig. 6. The performance of RPC with various cache size, normalized to the 1x mode refresh scheme.

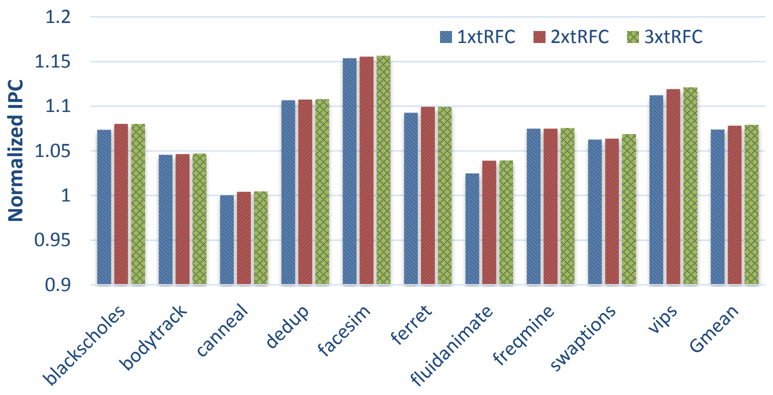
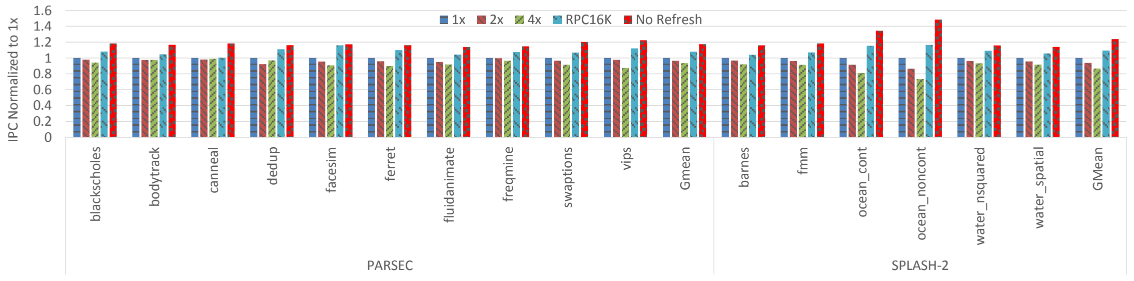


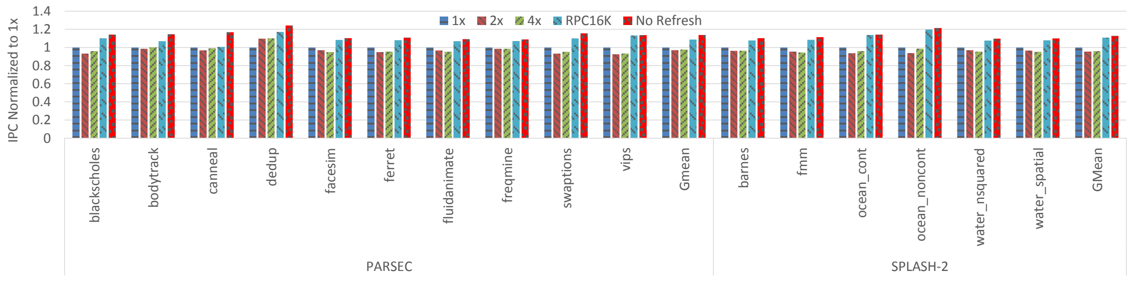
Fig. 7. The performance of RPC with various tCI, normalized to the 1x mode refresh scheme.

2.4.3 Single-Core Simulation Results

With the optimal cache size and the length of the caching interval (tCI), we next compare RPC to FGR and No Refresh schemes. All benchmarks are run on a single-core and single-thread system, and the evaluation results with regard to the different number of ranks are shown in Figure 8. It is clear that for all benchmarks



(a) 16GB two-rank memory system.



(b) 32GB four-rank memory system.

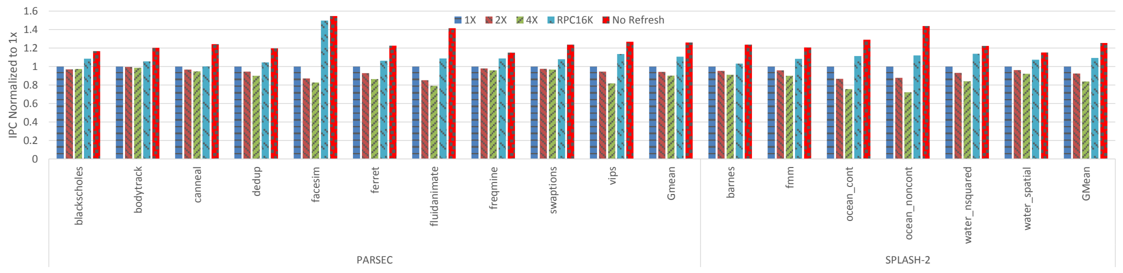
Fig. 8. Performance comparisons among *FGR*, *RPC* and *No Refresh* schemes with various number of ranks in a single-core system. *RPC* outperforms all *FGR* modes and improves system performance by 8.1% (8.7%) and 9.6% (10.8%) on average for PARSEC and SPLASH-2 benchmark suites in the two-rank (four-rank) system, respectively.

there is a large performance gap between 1x mode and No Refresh. Compared to *No Refresh*, the geometric mean of performance degradation is 17.4% (PARSEC) and 23.9% (SPLASH-2) for a two-rank system, and 13.7% (PARSEC) and 12.7% (SPLASH-2) for a four-rank system. In particular, the performance degradation of *ocean_noncont* benchmark is up to 48.5%. From these results we have two observations. First, most benchmarks in these two benchmark suites are memory-intensive workloads and memory system design has a huge impact on application performance. Second, conflicts between accesses and refreshes are reduced as the number of rank increases, because less memory requests go to each rank comparing to two-rank systems. As a result, the performance of 2x and 4x modes are worse than that of the 1x mode in most cases. And the performance of 2x and 4x modes in the four-rank system is better than that in the two-rank system.

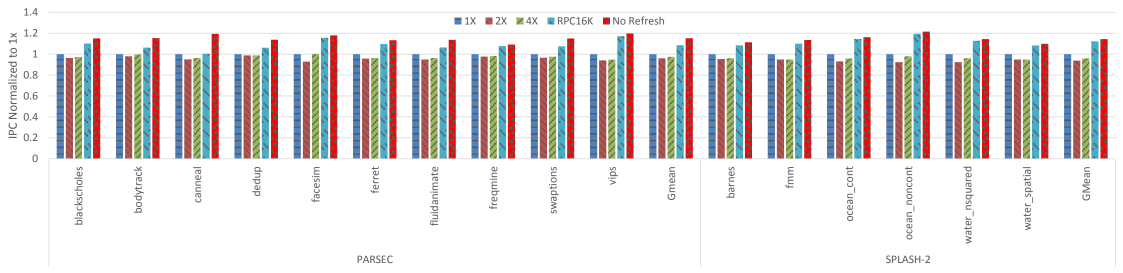
It is shown in Figure 8 that *RPC* outperforms all *FGR* modes, because it can serve memory requests and refreshes concurrently while *FGR* becomes ineffective for the memory-intensive workloads as discussed in Section 2.2.3. *RPC* improves the system performance by 8.1% (8.7%) and 9.6% (10.8%) on average for PARSEC and SPLASH-2 benchmark suites in the two-rank (four-rank) system, respectively. And *RPC* is comparable to the ideal case (around 95% of No Refresh). However, the performance improvement of *canneal* is less than 1% due to the poor temporal locality as shown in Figure 5a.

2.4.4 Four-Core Simulation Results

All runs are repeated on a four-core system with one thread per core to gauge the effectiveness in a multi-core environment, and the evaluation results with regard to the different number of ranks are shown in Figure 9. Since memory access becomes more intensive in these runs, comparing 1x mode to No Refresh, the performance degradation increases to 26.9% (15.2%) and 25.5% (14.4%) on average for PARSEC and SPLASH-2 benchmark suites in the two-rank (four-rank) system, respectively. On average, the performance of both 2x mode and 4x mode are worse than the 1x mode as expected. In contrast, *RPC* improves the system performance by 10.7% (8.6%) and 9.3% (12.2%) on average for these two benchmark suites in the two-rank (four-rank) system, respectively. And *RPC* can still achieve around 93% performance of No Refresh scheme, thus *RPC* scales very well in the multi-core environments.



(a) 16GB two-rank memory system.



(b) 32GB four-rank memory system.

Fig. 9. Performance comparisons among *FGR*, *RPC* and *No Refresh* schemes with various number of ranks in a four-core system. *RPC* outperforms all *FGR* modes and improves system performance by 10.7% (8.6%) and 9.3% (12.2%) on average for PARSEC and SPLASH-2 benchmark suites in the two-rank (four-rank) system, respectively.

2.5 Related Work

Refresh Reduction. Ghosh et al. [31] proposed *Smart Refresh* to eliminate unnecessary refresh operations. It leverages the characteristics that a read/write is equivalent to refresh due to the destructive access. Each row is bounded to a counter which gets reset whenever the row gets read out or written to. However, *Smart Refresh* requires very high storage overhead in the memory controller (e.g. up to 1.5MB in a 32GB memory system) [32], [33], and its effectiveness depends on the working set. *RAIDR* [32] proposed by Liu et al. also aims to reduce the number of refreshes. *RAIDR* uses the knowledge of cell retention times to group DRAM rows into retention bins and applies different refresh rates to different bins. As a result, rows containing leaky cells are refreshed at a normal rate, while most rows are refreshed less frequently. However, this retention-aware approach requires an accurate retention time profile which is hard to be determined due to the *Variable Retention Time (VRT)* [34, 35, 36].

Some software solutions were also devised to reduce refreshes. *RAPID* proposed by Venkatesan et al. [37] exploits retention time variations among different DRAM cells. The pages with longer retention time are allocated with priority over those with shorter retention time. The refresh rate is determined by the page with the shortest retention time among all allocated pages. However, *RAPID* has the same risk as *RAIDR* due to the variation in retention time, which may cause data reliability issue. In addition, its effectiveness depends on the utilization of the memory pages. *Flikker* [38] is another software solution to save refresh power by reducing the number of refreshes. In *Flikker* system, data is divided into critical and non-critical data. The portion of memory containing critical data is refreshed at the regular refresh rate, while the other is refreshed at a much lower rate to save power, which inevitably leads to retention errors. However, *Flikker* requires substantial modifications (to application, OS, DRAM chips) to implement it. All of the above mentioned refresh reduction solutions are orthogonal to our approach.

Refresh Pausing. Nair et al. [18] proposed *Refresh Pausing* to alleviate refresh overhead by allowing refresh operations to be interruptible. As a result, memory requests arriving during the refresh period can be serviced in a timely manner via pausing the on-going refresh operation. However, refresh operations have to be paused and resumed very frequently under memory-intensive workloads. In addition, the refresh operations become uninterruptible if DRAM rows are refreshed in a staggered or pipelined way [33].

Refresh Scheduling. *Elastic Refresh (ER)* [1] is proposed to mitigate refresh overhead. It leverages the refresh scheduling flexibility: up to eight refresh commands can be postponed. The basic idea is to avoid interferences between DRAM refreshes and memory requests that come outs during refresh periods by postponing the refresh commands for a predicted time period, which is based on the average idle period of a rank and the number of postponed refresh commands. In contrast to DUE [24], *ER* defers the refresh commands even when the to-be-refreshed rank is idle. However, *ER* becomes less effective under memory-intensive workloads since the average rank idle period is too short to hide the refresh period. Moreover, *ER* can adversely incur extra delay when it incorrectly predicts a time period as idle as discussed in Section 2.2.3.

Mukundan et al. [33] propose *Delayed Command Expansion* (DCE) and *Preemptive Command Drain* (PCD) respectively, to mitigate refresh overhead. DCE intentionally withholds admission of memory requests into the command queue if the target rank is being refreshed to prevent them from wasting command queue resources. PCD prioritizes commands that map to the to-be-refreshed rank to drain these commands before the rank is refreshed. In doing so, PCD can make more room for other commands that map to other ranks, thereby increasing parallelism among ranks during refresh period.

Concurrent Refresh. Chang et al. [25] and Zhang et al. [23] proposed concurrent refresh mechanisms, both of which increase the refresh granularity to a subarray so that refreshes and accesses can be serviced concurrently in different subarrays in the same bank. *RPC* distinguishes itself from those concurrent architectures in the following respects. First, our solution does not rely on the underlying DRAM organization and can be applied to a broad category of DRAM organizations. Second, it is hard to eliminate conflicts completely between accesses and refreshes as each subarray contains dozens of rows. Bursty conflicts might happen when workloads have strong locality as shown in Section 2.3.3. In contrast, depending on the caching policy and cache size, RPC is able to reduce the probability of conflict between memory access and refresh to a lower level.

CHAPTER 3

ARCHITECTING DIE-STACKED DRAM AS A PART OF MEMORY

3.1 Introduction

Recent advances in die-stacking technology have made it possible to integrate a large amount of DRAM in the same package of a processor. A processor and on-chip DRAM are interconnected by a high-density, low-latency through-silicon vias (TSVs). This technology has the potential to overcome the memory wall problem [3] by providing an order of magnitude higher bandwidth and much lower latency for on-chip DRAM. Prior work [39, 40, 41, 42, 43, 44] has proposed using die-stacked DRAM as a hardware-managed last-level cache (i.e., DRAM cache). As the technology for manufacturing die-stacked DRAM matures, the size of die-stacked DRAM could be tens of gigabytes by integrating multiple DRAM stacks on a 2.5D interposer [45, 46]. Therefore, using die-stacked DRAM as a DRAM cache would squander a large fraction of total memory space as the DRAM cache is invisible to the OS. Without fully exploiting the memory capacity offered, applications with a large working set would suffer a higher rate of page faults and therefore slowdown due to frequent accesses to backend storage.

An alternative to using die-stacked DRAM as a cache is to use it as part of an OS-visible memory space (i.e., PoM). In such a heterogeneous memory system, data residing in on-chip DRAM is serviced at high bandwidth and low latency, while data residing in off-chip DRAM is serviced at low bandwidth and high latency. However, naively treating on-chip DRAM as a part of memory space renders the PoM design less effective. To obtain high performance, on-chip DRAM needs to play two roles at the same time in a PoM architecture. The on-chip DRAM is not only a part of memory space but also a cache for off-chip DRAM. In other words, requested data is swapped or migrated to on-chip DRAM and victim data is swapped out to off-chip DRAM. This swapping process can be done by either the OS or hardware. For OS-managed approaches, the OS needs to monitor all page usage and migrate hot pages to the on-chip DRAM. OS-invoked page migrations result in page table updates and TLB shoot-downs, which are costly operations. Therefore, the page migrations under the OS control cannot occur frequently so that hot pages in a short period of time could not be migrated to the on-chip DRAM, resulting in performance loss. In contrast, in a hardware-managed PoM architecture, the migration is transparent to OS, and can be initiated at anytime when data is required. Hence, the hardware-managed PoM is a promising design and we only consider hardware-managed PoM in this work.

Current PoM designs [47, 48] fall into two categories based on the granularity at which they swap data: line-based and page-based (or segment-based). The line-based design uses off-chip bandwidth efficiently as all swapped lines are demanded. However, the line-based design could suffer from low hit ratio due to poor temporal locality at the main memory layer as highly referenced cache lines have already been filtered out by L1 and L2 caches. The page-based design swaps data at a coarser granularity (typically 1-4KB), thus achieving a higher hit ratio by exploiting spatial locality in the large granularity. However, the page-based design would waste precious off-chip bandwidth as some lines may not be touched before they are swapped out. The inefficient usage of off-chip bandwidth could lead to performance degradation, especially for data-intensive applications.

To take advantage of both line-based and page-based PoM designs while avoiding their respective drawbacks, we propose SELF [49], a high performance and memory bandwidth efficient approach to using die-stacked DRAM as a part of memory. SELF only swaps those lines in a requested page that are likely to be accessed according to its page footprint. In doing so, SELF enables most incoming requests to be serviced from on-chip memory while avoiding swapping unused lines to save memory bandwidth.

3.2 Background and Motivation

As more cores are integrated into many-core chips to improve processing capabilities and parallelism, the growth in core count requires a commensurate increase in memory bandwidth. However, memory speeds have not kept pace with CPU performance scaling, which has led to the memory wall problem [3]. Die-stacked DRAM has been advocated as a promising technology to break the memory bandwidth and latency wall. It provides an order of magnitude higher bandwidth and lower access latency than off-chip DRAM due to the dense TSVs buses [50]. However, the capacity of die-stacked DRAM is insufficient to fully replace off-chip DRAM due to technological constraints [41, 48]. Thus, die-stacked DRAM and off-chip DRAM will co-exist in future systems, and die-stacked DRAM can be used either as a cache or as a part of main memory. Most prior work [40, 41, 42, 43, 44, 51, 52, 53] advocates using die-stacked DRAM as a giant cache between the last level cache (LLC) and main memory, and copes with challenges of tag storage overhead, hit ratio, hit/miss latency and off-chip traffic etc. However, DRAM cache is invisible to the OS. In other words, DRAM cache cannot contribute towards the main memory capacity, which could lead to non-negligible performance loss due to increased page faults, especially for modern server applications with a large working set size (WSS). As the technology for manufacturing die-stacked DRAM matures, the size of die-stacked DRAM in each package could be up to tens of gigabytes. In this case, using die-stacked DRAM as a cache could waste a large fraction of total memory space.

Therefore, researchers have proposed using die-stacked DRAM as a part of memory [48, 47, 54, 55] instead of a cache. However, we can only get marginal benefits if the die-stacked DRAM is naively treated as a part of memory [55]. To obtain high performance, highly referenced pages or lines need to be migrated to die-stacked DRAM to take advantage of its high bandwidth and low access latency. This migration process can be performed by the OS or hardware.

3.2.1 OS-managed PoM

OS-managed PoM approaches need to track page usage to identify highly referenced pages. For an on-chip DRAM with a capacity of N pages, the OS should choose the top-N most referenced pages and map them into the on-chip memory at run-time. However, the operating system has a limited capability to get such information from the page table as the reference bit in each page table entry (PTE) cannot differentiate which pages are most referenced. A typical solution is to use a counter in each PTE to record the number of LLC misses per page, which would require extra hardware support [56]. At the end of each epoch or interval (e.g. 100K cycles), the OS sorts pages based on the access count and migrates the top-N hottest pages which are resident in off-chip memory to the on-chip memory. At the same time, these pages which are resident in on-chip DRAM but not belonging to the top-N hottest pages are migrated back to off-chip DRAM. Then the OS has to update the page table to reflect new mappings and invalidate corresponding translation lookaside buffer (TLB) entries (i.e., TLB shoot-down) for consistency. Therefore, the data migration under the OS control results in high overhead of sorting, copying pages back and forth between on-chip and off-chip memories, and TLB shoot-downs. As such, OS-managed migration cannot be performed frequently, which could miss many opportunities to improve performance by migrating pages that are highly referenced in short periods of time. In addition, OS-managed data migration can only occur at a page granularity (typically 4KB). When a significant fraction of data lines are not referenced, such page granularity transfers become very inefficient in terms of off-chip memory bandwidth. In a word, OS-managed PoM approaches could neither exploit the full benefits of on-chip DRAM at a coarse-grained interval nor utilize the off-chip memory bandwidth efficiently at a page granularity.

3.2.2 Hardware-managed PoM

Hardware-managed PoM can avoid page table updates, TLB shoot-downs and page sorting by maintaining a hardware-managed remapping table, which records real locations after swapping. The remapping table is updated by hardware without involving the OS after each data migration completes. Hence, the data migrations under hardware control could occur whenever the requested data is not resident in the on-chip memory, which could potentially improve the system performance. According to swapping granularity, hardware-managed PoM designs fall into two categories: line-based and page-based.

3.2.2.1 Line-based PoM

The line-based design swaps data at a line granularity. The small line granularity ensures a low utilization of off-chip bandwidth, since all lines swapped into the on-chip memory are demanded without wasting off-chip bandwidth. However, the line-based design falls short of exploiting abundant spatial locality, and temporal locality at the main memory layer is usually very poor as it has already been filtered out by the L1 and L2 caches. As a result, the line-based design suffers from a high miss rate of on-chip memory, accessing off-chip memory with low bandwidth and long access latency frequently.

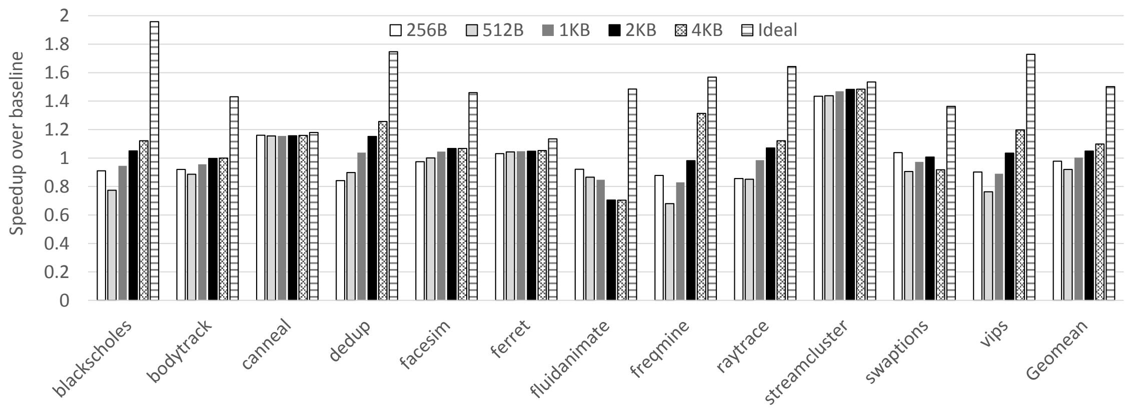


Fig. 10. The performance of a state-of-the-art page-based PoM design [47] with different page sizes. All requests are serviced from on-chip memory in the ideal case. On average, the page-based design performs best at the page size of 4KB.

3.2.2.2 Page-based PoM

The page-based design swaps data at a page (1-4KB) granularity. Compared to the line granularity, the large page granularity exploits abundant spatial locality, which could result in a higher hit ratio. Hence, the performance can be potentially improved as most misses in the LLC will likely be serviced from the on-chip memory at high memory bandwidth and low access latency. However, the large swap granularity could increase off-chip traffic as some unneeded lines of a swap-in page are also swapped in on-chip memory. The increase of off-chip traffic prolongs latency of off-chip accesses as the off-chip bandwidth is often overloaded, thus offsetting the benefit of hight hit ratio. Figure 10 shows the performance of state-of-the-art page-based PoM design while varying page size from 256B to 4KB. The results show that there is no one page size that can fit all cases. Smaller page sizes even degrade performance in some applications (e.g., *dedup*). On average, the page-based design performs best at the page size of 4KB. However, there is still a big performance gap between the best page-based design and the ideal case. The root cause is that the coarse page granularity cannot avoid wasting off-chip bandwidth, leading to saturation. In the case of off-chip bandwidth saturation, all requests to off-chip memory need to wait a long time in the transaction queue of memory controller, and are serviced sequentially, significantly degrading system performance.

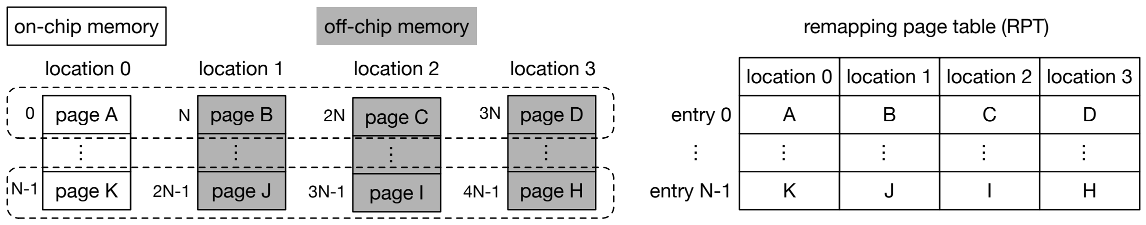
In conclusion, the line-based design uses off-chip memory bandwidth efficiently but suffers from a low hit ratio due to limited temporal locality. In contrast, the page-based design provides a higher hit ratio by exploiting spatial locality, while wasting off-chip bandwidth due to swapping useless data. To take advantages of both line-based and page-based designs while avoiding their drawbacks, we propose SELF, a high performance and bandwidth efficient approach to using on-chip DRAM as a part of memory.

3.3 Architecture and Design

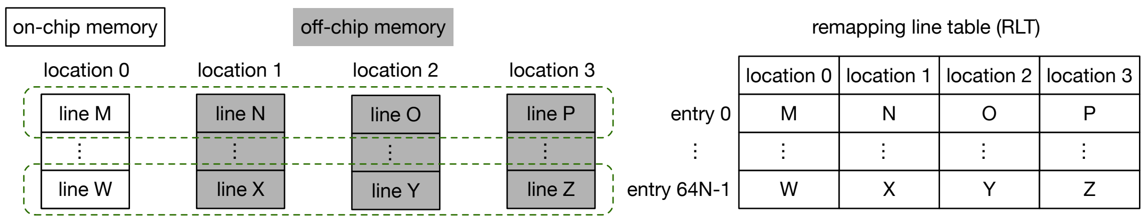
In order to gain similar high hit ratio as the page-based design while avoiding unnecessary off-chip traffic due to swapping useless data lines, we propose to selectively swap data lines of a page that are likely accessed during the page's residency in on-chip memory instead of blindly swapping an entire page. However, current remapping table of the page-based design does not support partial swapping as it cannot differentiate which data line has been swapped due to its page granularity.

3.3.1 Remapping Table Design

To achieve partial swapping of a page, we design two remapping tables at the granularity of page and line, called remapping page table (RPT) and remapping line table (RLT), respectively. The RPT is used to track pages' physical locations after swapping while the RLT records all data lines' physical locations in each page. In other words, the RPT tells where the requested page is and the RLT further indicates



(a) Direct page-remapping and remapping page table.



(b) Direct line-remapping and remapping line table.

Fig. 11. Direct remapping and corresponding remapping tables.

where the requested line is. With the cooperation of RPT and RLT, SELF can only swap those lines in a page that are likely accessed in the future to save off-chip bandwidth. In the PoM design, each LLC miss must first look up the remapping table to determine the actual physical location of the requested data. Then the memory controller can decide where to fetch the requested data from either on-chip memory or off-chip memory. In theory, data in the off-chip memory can be swapped to any location of on-chip memory in a similar way to a fully associative cache. In this case, we may need to search the entire remapping table in the worst case. As accessing the remapping table is on the critical path, searching the whole remapping table could cause excessive latency. To reduce the remapping table lookup time, we adopt direct-remapping, which is similar to the direct mapped concept in a cache design. In other words, a page or a data line in the off-chip memory can only be swapped to a specific location in the on-chip memory.

Figure 11 shows direct remapping applied in an example of memory system, in which the on-chip memory has a capacity of N pages, and the off-chip memory has

3N pages. Figure 11a shows direct page-remapping and corresponding RPT. Under the direct page-remapping, a page is only allowed to be swapped with another page mapped to the same entry of the RPT. For example, page A, page B, page C, page D are mapped to entry 0 of the RPT, thus they can be swapped with each other. Figure 11b shows direct line-remapping and corresponding RLT. It works in a similar way of the direct page-remapping. Due to the use of direct remapping, every remapping information can be retrieved with a single access to a corresponding entry. The RPT is indexed by the least significant log_2N bits of the requested physical page number (PPN) and the RLT is indexed by the least significant $log_2 64N$ bits of the requested line address. However, both RPT and RLT are on the critical path, each LLC miss needs to go through them sequentially. How to reduce or hide access latency of these two remapping tables plays an important role to the system performance. The RPT is small due to the use of coarse granularity. For the evaluated memory system consisting of 4GB on-chip DRAM and 12GB off-chip DRAM, the number of the RPTs entries is one million and each entry is a four elements tuple with two bits for each element. Therefore, the size of RPT is 1MB. However, the RPT could be more than ten megabytes as the on-chip DRAM keeps increasing. In order to be scalable and reduce access latency, we store the RPT in the on-chip memory and use a small SRAM (32KB), called RPT cache, to cache it. The RPT cache is indexed by the least significant log_2K bits of the physical page number, where K is the total number of sets in the RPT cache. And the least significant log_2N bits of the PPN is used as a tag. The RPT cache is expected to gain a high hit ratio because of a good spatial locality provided by the page granularity. In contrast, the RLT has a poor spatial locality due to the use of fine-grained granularity and it is very large (64MB in our evaluated system). Therefore, we choose to store the RLT in the on-chip memory only without caching it, which causes an extra access to the on-chip memory as each request must

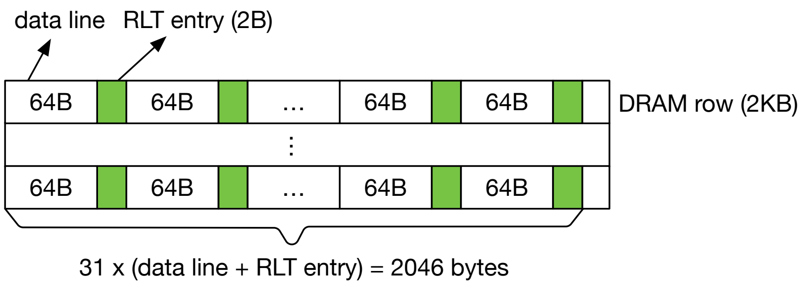


Fig. 12. The data layout of on-chip memory after co-locating each data line with its corresponding RLT entry.

first look up the RLT to determine the physical location of the requested line. To hide the access latency of RLT, SELF co-locates each data line with its corresponding RLT entry. This technique is also used in [40, 48]. To implement the co-located RLT, we sacrifice memory space of one data line in each 2KB DRAM row, and use it to store RLT entries for other 31 data lines. Thus, each RLT entry can have up to 2 bytes, leaving 2 bytes unused in each row. Figure 12 shows the data layout of on-chip memory after co-locating each data line with its corresponding RLT entry. In order to support the co-located RLT, we reserve 1/32 off-chip memory space. The reserved space could be used for data that will not be swapped. Therefore, for a requested line address X in on-chip memory, its actual physical address equals to $X + X/31$. In doing so, a data line and its corresponding RLT entry can be streamed out in one access. If the requested line is present in the on-chip memory by checking the RLT entry, we can directly use the data line just read out together with the RLT entry, without any extra access to the on-chip memory. If the RLT entry identifies that the requested line is in the off-chip memory, then a second access for the desired location in off-chip memory is performed.

3.3.2 Page Swapping

In the direct page-remapping, some pages (e.g., 4 pages in our system) are mapped to the same entry, and they compete for one location in the on-chip memory. When and which page should be swapped to the on-chip memory depends on the swapping policy. Ideally, the swapping policy should choose the hottest page in a certain period of time to be swapped to the on-chip memory, so that most incoming requests can be serviced from the on-chip memory. The most direct way is to record the number of accesses to each page during an interval by associating a counter with each page, then choose the page with the highest number of accesses to be swapped to the on-chip memory. However, the ideal swapping policy is too costly to implement in hardware. The simplest way is to swap the page to on-chip memory once it is demanded, which could cause frequent page swapping, especially when two pages mapped to the same entry are accessed in an interleaved fashion. As a result, the requested two pages are swapped back and forth, leading to saturating the off-chip bandwidth and wasted energy. In fact, pages residing in off-chip memory, called off-chip pages, are competing with the page residing in on-chip memory, called on-chip page. An off-chip page should be swapped to the on-chip memory as long as it is hotter than an on-chip page. Based on that, we employ a cost effective way by using a competing counter (CC) [47] to record the relative number of accesses. If the requested page is in the on-chip memory, the CC is decreased by 1, otherwise it is increased by 1. Once the CC is larger than the swap threshold, the off-chip page which is being accessed is swapped with the on-chip page. In our system the swap threshold is set to 8 (Section 3.4.7), so each CC only needs 4 bits. Due to address alignment, each CC is allocated 8 bits, some of which can be reserved for future use. To track page activity, each RPT entry is appended a CC.

3.3.3 Page Footprints

To achieve partial swap of a page, we need to predict which data lines in the page will be requested between two consecutive swap-in operations of the page and only swap those lines to reduce memory bandwidth consumption when a page swapping occurs. A lot of previous work [57, 58, 59, 60] demonstrate repetitive access patterns in commercial workloads. In other word, a data line that was accessed in current interval will likely be accessed in next interval. A page footprint records which data lines were accessed between two consecutive swap-in operations of the page. Based on that, we use page footprints to predict which lines in a page are likely accessed, which is similar to the Footprint Cache [41]. However, in main memory there is no tag array that can be used to record page footprints. Therefore, we redesign TLB to add a bit vector in each TLB entry to record a page footprint and also add a bit vector in each page table entry accordingly. The number of bits in a bit vector is equal to the page size divided by the data line size, thus each page footprint is typically 64 bits. If each core has a TLB of 32 entries, the storage cost of page footprints in the TLB is 256 bytes per core. The additional storage cost for page footprints in the page table is negligible since the page table is stored in main memory or disk. As all data requests have to lookup TLB, SELF can set the corresponding bits of the bit vector without extra accesses to the TLB, and the page footprint obtained from TLB can be directly used in the page swapping process without extra accesses to the page table either. Thus, recording page footprints in the TLB is a cost effective way.

However, recording page footprints in the TLB could cause incoherent problem in a multi-core system. As the page footprints' incoherency would not affect programs' correctness, we do not take any coherent action except any of these two cases occurs to reduce maintenance overhead of page footprints. 1) When a TLB entry is evicted, the page footprint from TLB bitwise OR with its corresponding page footprint in the page table and the result is saved in the page table but not synchronized with TLBs to improve prediction accuracy; 2) When a page is swapped to on-chip memory, its page footprints both in TLBs and the page table are reset to store the latest access information to reduce overpredictions (i.e. a data line is not requested but it was predicted). In either case, the related page table entry is updated. We update the page table through a system call to the page table walk. In the second case, the physical address is converted to a virtual address before the page table walk. We maintain a modified inverted page table to translate physical addresses to virtual addresses. Different virtual page mapped to the same physical page are stored in a linked list. Since updating the page table is not on the critical path and the page table can be accessed concurrently, the impacts of updating page table is negligible on the performance.

3.3.4 Line Location Prediction

As discussed in Section 3.3.1, we can save one access to the on-chip memory when the requested line is resident in on-chip memory by streaming the data line and RLT entry together. However, for the off-chip access (i.e., the requested line is resident in the off-chip memory), the RLT in the on-chip memory is accessed first to get the physical location of the requested line, then the off-chip memory is accessed according to the physical location. In this case, the off-chip access is serialized and occurs only after accessing on-chip memory. To break the serialized off-chip accesses, we reuse the RPT to predict line locations as the RPT itself has the information about page locations and most data lines in a page are likely to have the same location as its page. We use page locations obtained from the RPT to predict the locations of requested lines. If the line is predicted to be in off-chip memory, the predicted location in the

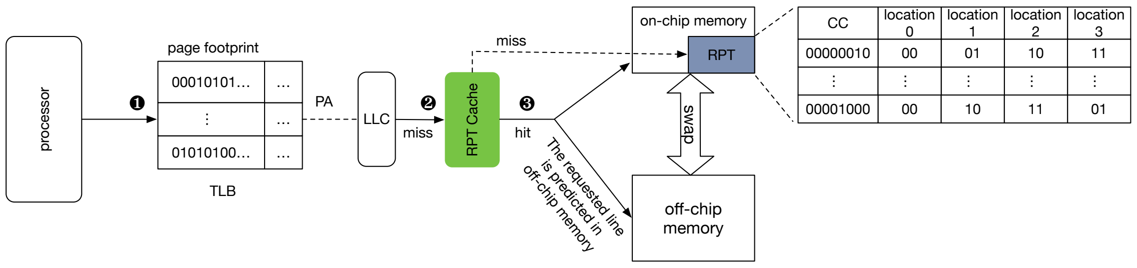


Fig. 13. Overview of SELF architecture. When the competing counter (CC) is larger than the swap threshold, SELF selectively swaps lines in the requested page according to its page footprint. Otherwise, SELF uses page location to predict the requested line location to reduce latency of off-chip accesses.

off-chip memory will be accessed in parallel with on-chip access. If the prediction is correct, the line from off-chip location is used and the latency of RLT access is hidden. If the requested line is found in on-chip memory by checking the RLT entry, then the prediction is ignored. In the worst case, the requested line is in off-chip but it is predicted to be in a wrong off-chip location, a second access to the off-chip location still need to be performed.

3.3.5 Put Everything Together

The SELF integrates all techniques presented in above sections, as shown in Figure 13, where the on-chip memory accounts for a quarter of the total capacity. For other ratios, SELF works similarly, but the storage overhead of the RPT and RLT may be slightly different. ❶ A request from the processor accesses the TLB to get its physical address (PA) of the requested data, and set corresponding bit in its page footprint at the same time. ❷ The RPT cache is accessed if the request is missed in the LLC. If the request is a cache miss, then a corresponding RPT entry is loaded to the RPT cache. Otherwise, a RPT entry related to the request is accessed.

According the real location of the requested page, the CC of the accessed RPT entry is updated. If the CC is larger than the swap threshold, SELF selectively swaps those lines of the requested page to on-chip memory according to its page footprint obtained from step ❶ and resets its CC and page footprint. Otherwise, SELF uses the page location to predict the requested line location. If the requested line is predicted in off-chip memory, the predicted location in off-chip will be accessed in parallel with on-chip access, or only on-chip access will be issued if the requested line is predicted in on-chip memory. ❸ The RLT entry and a data line are returned together from the on-chip memory. According to the RLT entry, if the real location of the requested line is in the on-chip memory, the data line is used to service the request directly and ignore any prediction. If the real location of the requested line is in the off-chip memory and was predicted correctly, memory controller only needs to wait until the requested data returned from the off-chip memory. In this case, latency of retrieving the RLT is avoided as it was issued in parallel with off-chip access to the predicted location in step ❷. However, if the real location of the requested line is in the off-chip memory and was wrongly predicted, then an access to the real location in the off-chip memory is performed.

In a word, all techniques applied in SELF work in concert to enable most incoming requests to be serviced from on-chip memory while avoiding swapping unused lines to save memory bandwidth. SELF also reduces latency of off-chip accesses by smartly reusing the RPT as a line location predictor.

3.3.6 Overhead Comparison

We compare the storage overhead of SELF with state-of-the-art line-based and page-based designs, called CAMEO [48] and PoM [47], respectively. Table 5 shows the storage overhead of these three designs under a memory system composed of

4GB on-chip DRAM and 12GB off-chip DRAM. In such a system, there are 1 million entries in the RPT. Each entry occupies 2 bytes (one byte is allocated to a CC and the other byte is used to store page locations). Thus, the storage overhead of the RPT is 2MB. Moreover, as discussed in Section 3.3.1, each DRAM row needs to sacrifice one data line out of 32 data lines to implement the co-located RLT. Therefore, the total storage overhead of RPT and RLT is 4GB/32 + 2MB = 130MB. Compared to CAMEO and PoM, first, SELF requires additional storage space in the TLB, 256 bytes per core, to record page footprints. Second, SELF needs more SRAM space than CAMEO. However, SELF could have higher prediction accuracy as CAMEO only uses 512 bytes to record last accessed locations and relies on them to predict requested line locations. Third, SELF consumes more space of on-chip DRAM than PoM and CAMEO, but it is still negligible, only 3.2% of the total capacity. In summary, SELF introduces more storage overhead than CAMEO and PoM, but it achieves two conflicting goals of high hit ratio of on-chip memory and low off-chip traffic.

Table 5. Storage Overhead Comparison.

Storage	CAMEO	PoM	SELF
TLB	N/A	N/A	256B/core
SRAM	512B	32KB	32KB
on-chip DRAM	128MB (3.1%)	2MB (0.05%)	130MB (3.2%)

3.4 Evaluation

3.4.1 Evaluation Methodology

We use a full system and cycle accurate simulator, MARSSx86 [30], with a detailed DRAM simulator, DRAMSim2 [26], for our evaluations. The DRAMSim2 is modified to support multiple memory instances. We use two instances of DRAMSim2 with different configurations [22] to model both on-chip DRAM and off-chip DRAM. The evaluated memory system consists of 4GB on-chip memory and 12GB off-chip memory. We use a system composed of 16GB off-chip DRAM without on-chip DRAM as our baseline system. Table 6 shows the system configuration in our study.

Table 6. System Configuration.

CPU	
Core	8 cores, 3.2GHz out-of-order, 4 issue width
L1-D/L1-I cache	8-way, 128KB/128KB, 2 cycles
L2 cache	8-way, private 1MB, 8 cycles
L3	16-way, shared 16MB, 24 cycles
RPT cache	4-way, 32KB, 2 cycles, LRU replacement
Die-stacked DRAM	
Bus frequency	1.6GHz (DDR 3.2GHz)
Channels/Ranks/Banks	8/1/8
Bus Width	128 bits per channel
tCAS-tRCD-tRP-tRAS	11-11-11-28
Off-chip DRAM	
Bus frequency	800MHz (DDR 1.6GHz)
Channels/Ranks/Banks	2/1/8
Bus Width	64 bits per channel
tCAS-tRCD-tRP-tRAS	11-11-11-28

We use the PARSEC 2.1 [27] benchmark suite to evaluate our design. PARSEC 2.1 includes emerging applications ranging from computer vision to financial analytics. And it is a multi-threaded and memory-sharing benchmark suite, thus it is suitable for evaluating memory system. Each benchmark of PARSEC has defined a range of interest (ROI) to represent the workload and we checkpoint each benchmark at the beginning of the ROI. We launch simulations from checkpoints with warmed caches and page footprints to achieve a steady state. Each benchmark runs for 500 million instructions with simlarge input dataset to collect statistical data.

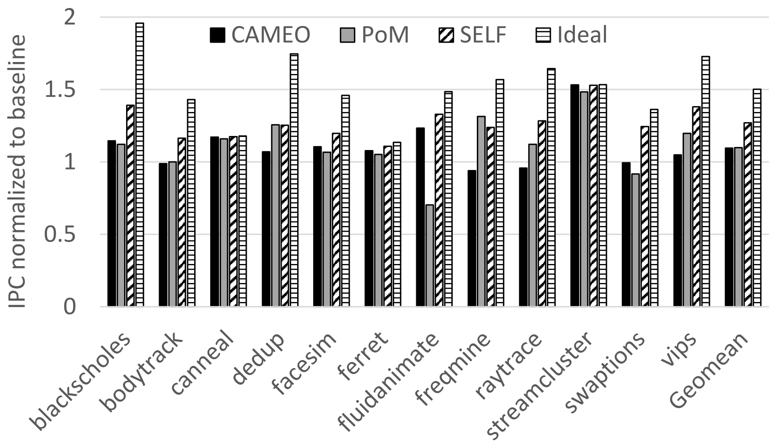


Fig. 14. Performance comparisons. On average, CAMEO and PoM improve performance by 9.5% and 9.9%, respectively, while SELF improves performance by 26.9%, which is 85% of the ideal case.

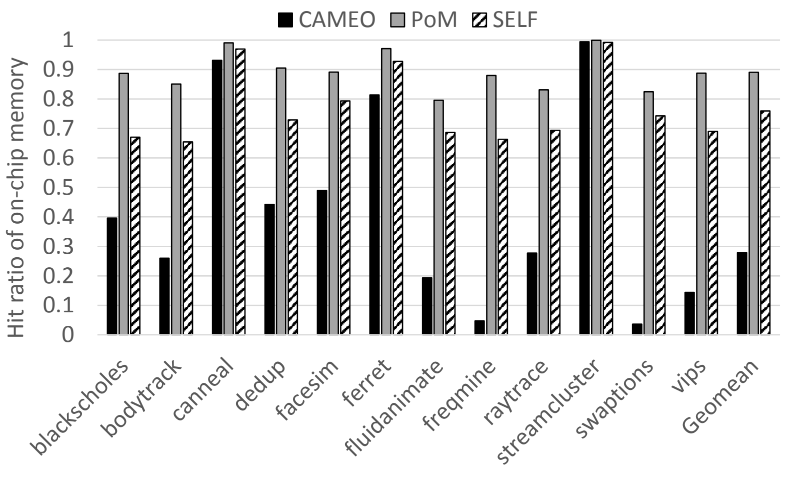
3.4.2 Performance Results

We compare SELF with CAMEO [48] and PoM [47]. We also compare these three designs against an ideal memory system composed of all on-chip DRAM without off-chip DRAM. The ideal memory system is also set to 16GB for fair comparison. We use instructions per cycle (IPC) as our performance metric. Figure 14 shows the performance results of various designs, which are normalized to the baseline system. On average, SELF improves performance by 26.9%, which is 85% of the ideal case. However, CAMEO and PoM improve performance by 9.5% and 9.9%, respectively. From the figure we can see CAMEO in some benchmarks, e.g., *freqmine* and *raytrace*, is even worse than the baseline system. There are two possible reasons for the surprising results. On the one hand, the temporal locality of these workloads is very poor, thus most requests are serviced from the off-chip memory. On the other hand, the line location predictor (LLP) used in CAMEO cannot work well with these workloads as the LLP simply uses last accessed location to predict the requested line location. The two aspects together cause the worse performance than the baseline system. PoM in some workloads, e.g., *fluidanimate*, also performs worse than the baseline system. The main reason is that these workloads have a poor spatial locality which causes PoM to exhibit a low hit ratio of on-chip memory although it swaps at a page granularity. In this case, the coarse swap granularity could easily saturate the off-chip bandwidth, leading to a long latency for off-chip accesses. However, SELF performs steadily in all benchmarks by combining all benefits from CAMEO and PoM while avoiding their shortcomings.

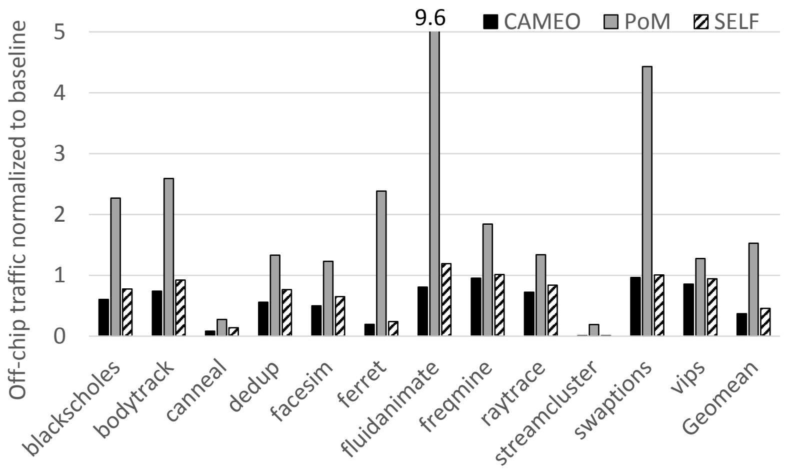
3.4.3 Hit Ratio and Off-chip Traffic

To further understand the above performance results, we collect data about two important performance metrics, hit ratio of on-chip memory and off-chip traffic, as shown in Figure 15. Figure 15a shows the percentage of requests serviced from the on-chip memory. Figure 15b shows how much data is read from the off-chip memory. We simulate write requests in the evaluation but write traffic is not calculated as write requests are not on the critical path. We compare SELF with CAMEO and PoM, all results are normalized to the baseline. For simplicity, we analyze these three designs respectively.

First, CAMEO gains the lowest hit ratio of on-chip memory among these three designs, only 28% on average. As CAMEO can only capture temporal locality due to the fine-grained line granularity used to swap data from off-chip memory to on-chip memory. Therefore, the strength of workloads' temporal locality decides the hit ratio of on-chip memory. The *freqmine* and *swaptions* have a very low hit ratio, less than 5%, thus most requests are serviced from off-chip memory. That is why their off-chip traffic is almost the same as the baseline. In general, CAMEO produces the least off-chip traffic, 37% of the baseline, as every data line read from the off-chip memory



(a) Percentage of requests serviced from the on-chip memory.



(b) The total data read from the off-chip meory.

Fig. 15. Two import performance metrics (a) hit ratio of on-chip memory and (b) off-chip traffic. All results are normalized to the baseline. SELF achieves an average hit ratio of 76% while reducing off-chip traffic to 46% of the baseline system. Although PoM obtains the highest hit ratio, 89% on average, it also causes the highest off-chip traffic, 153% on average.

are demanded although it gains the lowest hit ratio of on-chip memory.

Second, PoM design obtains the highest hit ratio of the fast memory, 89% on average. However, it also causes the highest off-chip traffic, 153% on average. Both high hit ratio and heavy off-chip traffic are attributed to the large page granularity used to swap data between on-chip and off-chip memories. Figure 15b shows *swaptions* and *fluidanimate* workloads have a very high off-chip traffic, especially for the *fluidanimate* workload, which causes almost an order of magnitude higher off-chip traffic than the baseline system. The high off-chip traffic could easily lead to saturating the off-chip bandwidth. As a result, requests missed in the on-chip memory need to wait for a very long time in the transaction queue of memory controller, which explains why *fluidanimate* gets the worst performance with PoM design, as shown in Figure 14.

Last, our proposed SELF achieves an average hit ratio of 76%, which is close to PoM design, and meanwhile reduces the off-chip traffic to 46% of the baseline system. The high hit ratio indicates page footprint has a low rate of underprediction, i.e., a data line is demanded but it was not predicted. In contrast, the low off-chip traffic indicates page footprint has a low rate of overprediction, i.e., a data line is not demanded but it was predicted. Therefore, the page footprint is a good predictor for a page's spatial pattern. SELF takes advantage of page footprint to do partial swapping of a page to achieve both high hit ratio of on-chip memory and low off-chip traffic. Moreover, SELF reuses the RPT to predict line locations of off-chip accesses to shorten access latency. These techniques together ensures that SELF outperforms CAMEO and PoM designs on average.

3.4.4 Prediction Accuracy

In SELF, we reuse the RPT to predict the requested line location based on the location of page which the requested line belongs to. For assessing the accuracy of RPT, we first describe five possible cases that can occur: 1) the requested line is in the on-chip memory and the RPT predicted correctly; 2) the requested line is in the on-chip memory but the RPT predicted wrong; 3) the requested line is in the off-chip memory but it was predicted in the on-chip memory; 4) the requested line is resident in the off-chip memory and the RPT gave a right off-chip location; 5) the requested line is resident in the off-chip memory but the RPT gave a wrong off-chip location. Case 2, 3 and 5 are mispredicted cases, but have different misprediction penalties. In case 2, the request can still be serviced from the on-chip memory quickly, but energy and off-chip bandwidth consumed by off-chip access is wasted. In case 3, off-chip access is performed after on-chip access, increasing access latency. The penalty of case 5 is the sum of case 2 and 3. Table 7 shows the breakdown of prediction accuracy. In summary, the RPT achieves an average accuracy of 85.5% across all workloads.

Table 7. The breakdown of prediction accuracy.

Served by	Prediction	Percentage
On-chip	On-chip	75.5%
	Off-chip	4.1%
Off-chip	On-chip	4.8%
	Off-chip (right)	10%
	Off-chip (wrong)	5.6%
Overal Accuracy		85.5%

3.4.5 Energy Analysis

Figure 16 compares various designs in terms of energy per access. The energy per access is defined as total energy consumed by on-chip and off-chip memories without considering peripheral wires divided by the number of read and write requests. The results are normalized to the baseline system. We calculate power consumption based on the Micron Power Calculator [61] and the Micron DDR3 data sheet [22]. We modify

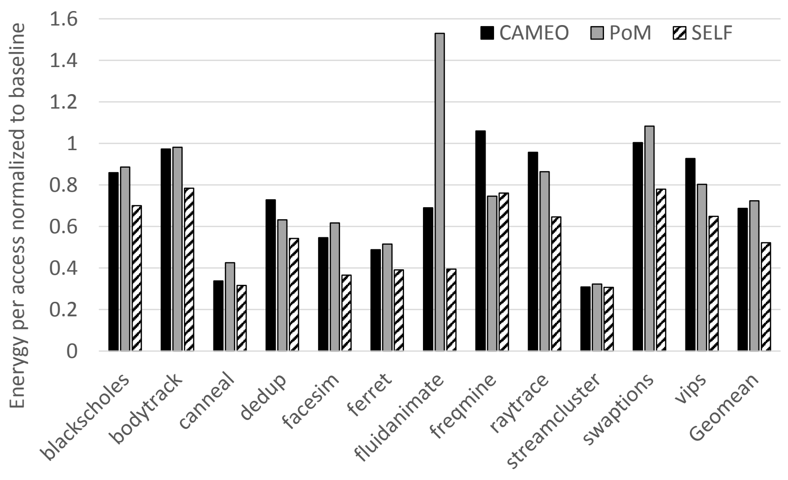


Fig. 16. Energy consumption. On average, CAMEO, PoM and SELF reduce energy per access by 31.3%, 27.6% and 47.9%, respectively.

power parameters according to the power number reported in [62] for the on-chip memory. The results show that CAMEO, PoM and SELF reduce energy consumption by 31.3%, 27.6% and 47.9%, respectively. Comparing these designs, CAMEO has zero overprediction while PoM has the most overpredictions among these three designs. Each overprediction wastes energy and increases off-chip traffic. The increased off-chip traffic may prolong the execution time, resulting in more static energy consumption. Therefore, PoM consumes more energy than CAMEO on average, especially for the *fluidanimate* workload as it causes the highest off-chip traffic, as shown in Figure 15b. Running the *freqmine* workload, CAMEO even consumes more energy than the baseline system. As this workload gets a very low hit rate of on-chip memory, most requests still need to access the off-chip memory to get the requested data after first accessing the on-chip memory, which causes a lot of extra on-chip accesses, compared to the baseline system where each request only requires one off-chip access. These extra on-chip accesses cause CAMEO to consume more energy than the baseline

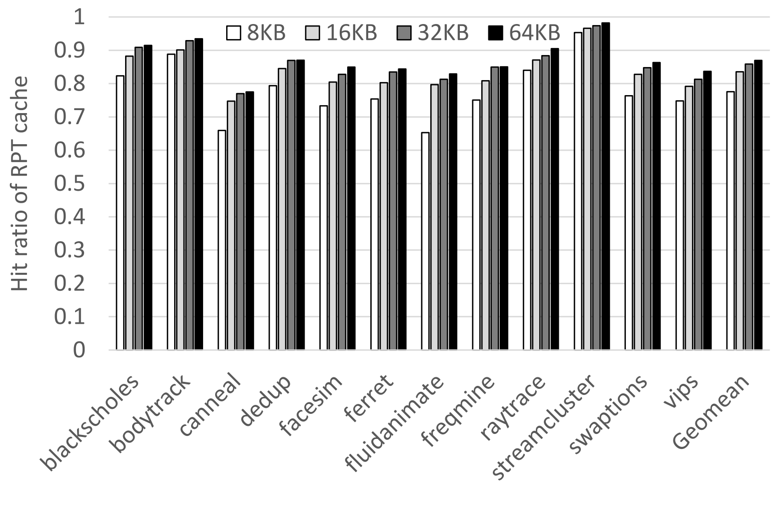


Fig. 17. Hit ratio of RPT cache across different cache size.

system. However, SELF reduces energy consumption across all workloads due to its high hit ratio of on-chip memory and low off-chip traffic.

3.4.6 Sensitivity to RPT Cache Size

We adopt a RPT cache to shorten access latency of the RPT. If a request hit in the RPT cache, the requested page location can be obtained quickly. Otherwise, the request needs to access the on-chip memory instead to get a corresponding RPT entry, which causes a much longer latency. Therefore, the effectiveness of the RPT cache is crucial to the system performance. Figure 17 shows the hit ratio of the RPT cache when we change its size from 8KB to 64KB. From the figure, all workloads can get a high hit ratio even with a 8KB RPT cache. On average, the hit ratio is 77.5%, 83.5%, 85.8% and 87%, respectively. The hit ratio of the RPT cache can be improved marginally by increasing the cache size after the RPT cache reaches 32KB. Thus, we choose 32KB as the RPT cache size.

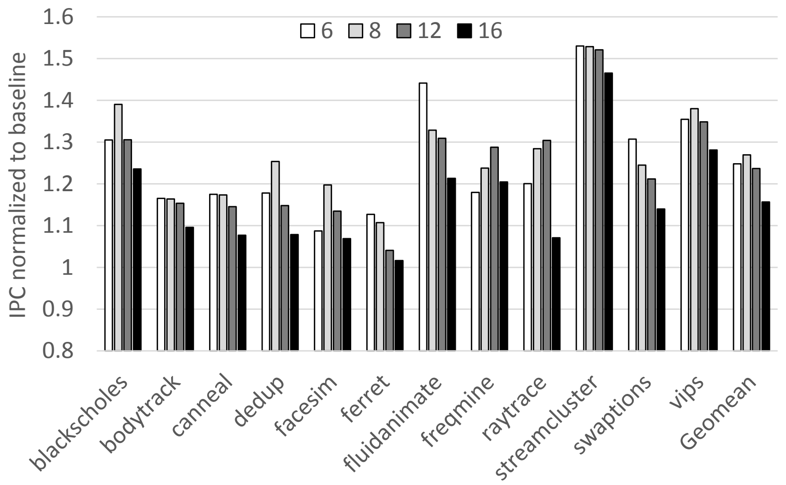


Fig. 18. Performance sensitivity to various swap thresholds (6, 8, 12, and 16).

3.4.7 Sensitivity to Swap Threshold

In SELF system, page swapping occurs when the CC is larger than the swap threshold. Thus, the swap threshold is closely related to the system performance. We perform a sensitivity study of swap threshold on the system performance and Figure 18 shows the results. There is no one swap threshold that can fit all workloads. For example, *ferret* and *swaptions* prefer small swap threshold as their spatial localities are strong. The earlier requested page is swapped to the on-chip memory, the better performance SELF can achieve. In contrast, for *freqmine* and *raytrace* benchmarks, the performance is improved as the swap threshold increases. However, the performance becomes worse in most workloads when the swap threshold is set to 16. The high swap threshold could make most pages have no chance to be swapped to on-chip memory, causing most requests to be serviced from off-chip memory. In this case, SELF could be degraded to the baseline system. For example, the performance of *ferret* is close to the baseline system when the swap threshold is set 16. Therefore, the ideal value of swap threshold should be configured according to the access pattern of each workload. However, we set it to 8 as a good trade-off since SELF achieves the best performance on average in this case. To improve the adaptability of swap threshold in the future, we are trying to use several small regions in on-chip memory as sampling regions and apply different swap thresholds for them. The swap threshold which can produce the highest benefit in current interval is adopted for the next interval. In doing this, the swap threshold is changed dynamically to adapt to the access pattern.

3.5 Related Work

DRAM Cache. A large body of previous work [39, 40, 41, 42, 43, 44, 51, 52, 53, 63] has proposed using on-chip DRAM as a hardware-managed cache between the LLC and main memory. DRAM caches can also be classified into two categories by caching granularity: line-based and page-based. These two categories have the same problems stated in this work. To alleviate the over-fetching problem of the page-based design, Footprint Cache [41] and Unison Cache [42] use a footprint predictor to identify and fetch only those lines within a page that will be requested during the page's residency in the DRAM cache. In doing so, they eliminate the excessive off-chip traffic associated with page-based cache designs, while preserving their high hit ratio. They are similar to our work but the on-chip DRAM is used as a cache and the tag array provides sufficient information about page footprints while this kind of information is missing in the main memory layer.

Part-of-Memory (PoM). DRAM cache has the advantage of being transparent to the OS. However, DRAM cache cannot contribute towards capacity of main memory, which could lead to non-negligible performance loss. Thus, many researchers advocate using die-stacked DRAM as a part of memory. Some hybrid approaches managed by both software and hardware have been proposed besides the hardware-managed PoM designs. Meswani et al. [54] propose the *first-touch hot-page (FTHP)* approach to managing a heterogeneous memory architecture (HMA). This approach needs support from both hardware and software. An access count is added to each TLB and page table entry to track the number of page accesses. At the end of an epoch, all pages whose access count is larger than the hotness threshold θ are treated as hot pages. The OS selects first N (N is the size of the die-stacked DRAM) hot pages to place in the stacked memory and updates corresponding PTEs. If the number of hot pages is more than N, the OS increases the hotness threshold, otherwise decreases it. In the case when the size of hot pages is less than N, the OS adopts first-touch policy to allocate requested pages in the stacked memory until it is used up. Although this approach makes use of hardware to fasten page profiling, the page table updates and TLB shoot-downs handled by the OS are still very costly, as discussed in Section 3.2.1. Thereby, page migrations cannot happen so frequently that many opportunities to improve performance could be missed.

Oskin et al. [55] propose a software-managed and hardware-assisted approach to use die-stacked DRAM as a part of memory. This approach leverages two techniques to make it be feasible. The first is a hardware-assisted TLB shoot-down to accelerate this process; the second is a software-implemented prefetcher that extends classic hardware prefetching algorithms to the page level. This approach requires simpler hardware than our approach, however, it performs data migration between on-chip and off-chip memories at a granularity of page, resulting in waste of the off-chip memory bandwidth.

CHAPTER 4

PAGE PLACEMENT IN DIE-STACKED DRAM/NVM MEMORY SYSTEMS

4.1 Introduction

The demand for memory capacity and bandwidth keeps increasing, which is mainly driven by the growing memory requirements of new applications, and the increasing number of processing cores in a single chip. However, the conventional DRAM-based memory systems cannot meet these needs due to scalability and power issues. First, the bandwidth of DRAM has not kept pace with processor scaling, which has led to the memory wall problem [3]. Second, DRAM dissipates considerable power and has been reported to account for as much as 40% of the power consumed by a high-end server [4]. The problem exacerbates with the increasing DRAM capacities, making such servers less energy proportional. Fortunately, emerging memory technologies provide some desired features, such as high performance, high density, and low power, but there is no single memory technology that owns all desired features. Thus, a hybrid architecture could be a promising way to build a high performance, large capacity, and energy efficient memory system.

To achieve this goal, we combine high bandwidth memory (HBM) and phase change memory (PCM) to constitute a hybrid memory system as they have complementary features. HBM [46] is a type of die-stacked DRAM and has the potential to overcome the memory wall problem by providing an order of magnitude higher bandwidth and lower latency than conventional DRAM, but its capacity is limited (currently several gigabytes). PCM is a byte-addressable non-volatile memory (NVM), thus it has near-zero standby power. Compared to conventional DRAM, PCM can offer high density, but it has longer access latency (∼2x for reads and 8x-16x for writes [64]) and limited write endurance. Due to their limitations, neither HBM nor PCM can replace conventional DRAM solely as a main memory. Thus, we combine them to form a hybrid memory system to take advantage of HBM and PCM while avoiding their disadvantages as many as possible. In order to fully exploit high performance (i.e. high bandwidth and low latency) offered by HBM and large capacity offered by PCM, hot pages should be migrated to HBM to improve performance and cold pages should be stored in PCM to save energy. Therefore, how to identify hot pages is very critical. A lot of work regarding DRAM/NVM hybrid memory systems [7, 8, 9, 10] has been proposed. RaPP [9] adopts a modified multi-queue [65] to profile page access and migrate top ranked pages to DRAM. RaPP is demonstrated to be effective to identify hot pages. However, it requires a sophisticated memory controller to maintain the Multi-Queue structure. HSCC [10] tracks page access via extending page table and translation lookaside buffer (TLB) and fetches those NVM pages whose access counts become larger than a given threshold into DRAM. Tracking page access in TLB incurs synchronization cost due to page sharing between different cores, especially when the number of cores is high. In summary, existing solutions track page hotness via redesigning memory controller or extending TLB. Therefore, they can be costly to be implemented. Moreover, the hot page migration only occurs when the number of page accesses exceeds the threshold, which could miss lots of opportunities to improve performance.

We propose a cost-effective and energy-efficient architecture for die-stacked DRAM /NVM memory systems, especially for HBM/PCM memory systems, called Dual Role HBM (DR-HBM) [66]. In DR-HBM, HBM plays two roles and is divided into two parts. A small portion of which, called HBM cache, is used as a cache for PCM.

The remaining HBM and PCM together constitute the main memory. In our design, the HBM cache has two purposes. On one hand, the HBM cache is used to bridge performance gap between the last level cache (LLC) and PCM, and absorb write requests to prolong PCM's lifetime. On the other hand, the HBM cache is also used to track page access. A page whose access count is higher than the migration threshold is going to be migrated to the HBM. In doing so, hot pages reside in the HBM cache before migration and then reside in HBM after migration. As a result, most requests will be serviced from the fast memory including HBM and the HBM cache, improving system performance1. Moreover, we propose three techniques to improve performance further and reduce writes to the PCM. First, CSM (cache on the second mis) increases the effectiveness of HBM cache and reduces PCM traffic by avoiding to cache singleton pages that contain only single useful data blocks; Second, hot pages are migrated to HBM in batches to amortize TLB shoot-down overhead; Last, we propose Hot First LRU (HF-LRU) page replacement policy and increase the weight of write operation to reduce writes to the PCM. The experimental results show that DR-HBM outperforms two state-of-the-art hybrid memory systems, called RaPP [9] and CAMEO [48], respectively. Compared to the baseline without page management, DR-HBM improves the performance by 63% while reducing energy consumption by 32.9% on average.

4.2 Background and Motivation

4.2.1 Emerging Memory Technologies

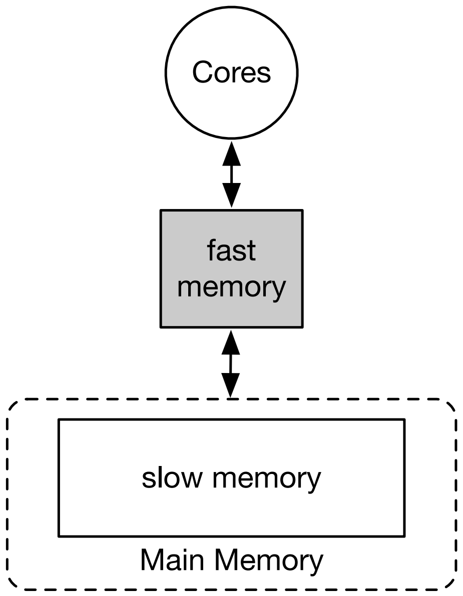
Die-stacked DRAM is a new memory technology where multiple DRAM dies are stacked vertically to form a DRAM stack. DRAM dies in each DRAM stack are connected by high-density, low-latency through-silicon vias (TSVs) [67]. A DRAM stack can be stacked on top of or next to ("2.5D stacking") a processing chip. By integrating multiple stacks on a 2.5D interposer, the capacity of die-stacked DRAM could be more than ten gigabytes. However, it is still insufficient to fully replace conventional DRAM [41, 48]. As die-stacked DRAM are integrated in the same package with the processor, avoiding the conventional pin-count limits on both the memory and processor packages, die-stacked DRAM can provide an order of magnitude higher bandwidth and lower latency than conventional off-package DRAM. For example, a single DDR3 channel clocked at 1600 MHz can provide a peak bandwidth of 12.8GB/s. Thus, a typical memory system that is equipped with 2-4 channels can provide a bandwidth of 25.6GB/s-51.2GB/s. While a single stack of die-stacked DRAM with eight channels, each of which has 128 bits at a data transfer speed of 1Gbps, can provide a peak bandwidth of 128GB/s in total. Similarly, a stacked memory system that includes 4 stacks can offer a bandwidth of 512GB/s. As the die-stacking technology becomes mature, die-stacked memory systems could provide even higher bandwidth by integrating more DRAM stacks in the on-chip package. Therefore, die-stacked DRAM technology has been widely embraced by industry as a viable solution to the "Memory Wall"problem [3]. Multiple industry standards such as High Bandwidth Memory (HBM) [46] and Hybrid Memory Cube (HMC) [45] have emerged to support this technology. In this work, we choose HBM as a representative of die-stacked DRAM.

As DRAM-based memory system is consuming an increasing proportion of the power budget, power consumption becomes a major concern while scaling. Non-volatile memory, such as PCM, is attracting more attention as a promising candidate for next generation memories [7, 8, 68, 69, 70, 71, 6]. PCM has the same organization as DRAM [6], but is a type of non-volatile memory. A PCM-based memory system has one or more memory controllers, each of which manages one or more channels. Each channel is composed of several ranks (1-4). A rank is a collection of PCM chips that together feed the 64-bit data bus. A rank is typically partitioned into 8 banks. Each bank consists of a two-dimensional array of PCM cells. A PCM cell consists of an access transistor and a storage resistor made of a chalcogenide alloy. With the application of heat, the alloy can be switched between two states, amorphous and polycrystalline. The amorphous phase has high resistance, whereas polycrystalline phase has low resistance. The difference in resistivity between the two states can be three to five orders of magnitude. Thus, some intermediate resistances that are achieved by controlling the proportion of the two states in a PCM cell have made it possible to store multiple bits per cell (MLC) [72]. The data stored in the cell is retrieved by sensing the alloy's resistance by applying very low power. Compared to conventional DRAM, PCM can offer higher capacity, especially for MLC PCM, and much lower static power. However, PCM has longer access latency (about 2x for reads and 8x-16x for writes) than DRAM and limited write endurance. These two drawbacks hinder PCM from being a replacement of DRAM.

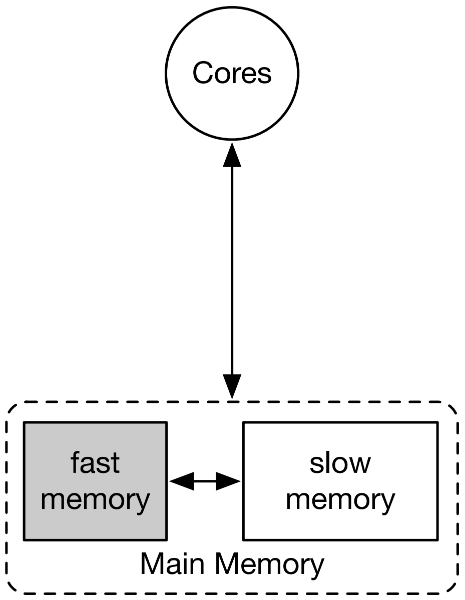
In summary, both die-stacked DRAM and PCM cannot fully replace conventional DRAM separately due to their own limitations. However, there is clear incentive for combining these two technologies into a hybrid memory system as they have complementary features.

4.2.2 Hybrid Memory Systems

There have been a number of research efforts on managing and architecting hybrid/heterogeneous memory systems [48, 47, 54, 7, 8, 9, 73, 74, 10, 75, 76, 77]. Most of them are comprised of two memory technologies with different characteristics. For generality, the memory that has lower access latency but smaller capacity is called fast memory. On the contrary, the memory that has higher access latency but larger capacity is called slow memory. The fast memory can be organized in two ways.



(a) Hierarchical hybrid memory architecture.



(b) Flat-addressable hybrid memory architecture.

Fig. 19. Typical hybrid memory system architectures.

One way is to architect the fast memory as a cache/buffer for the slow memory, as shown in Figure 19a. In this hierarchical architecture, the fast memory is a hardware-managed cache and it is transparent to OS. Therefore, applications can run without being modified. This architecture is good for a small fast memory (up to hundreds of megabytes). When the fast memory becomes larger, such as several gigabytes, the system loses a non-negligible portion of main memory space as the fast memory cannot contribute to overall main memory capacity, leading to a higher rate of page fault for capacity-constrained workloads and degrade system performance.

The other way is to organize the fast and slow memories together as a flat-addressable main memory, as shown in Figure 19b. Both the fast and slow memories are visible to the OS. For such system organization, the main challenge is to intelligently place and migrate data between the different memories to ensure optimal performance and energy efficiency. CAMEO [48] migrates requested data lines to the fast memory on demand and swap victim lines out to the slow memory. In other words, data swap occurs once the requested data is not in the fast memory. This approach is originally designed for the hybrid memory system composed of die-stacked DRAM and off-package DRAM (i.e. conventional DRAM). The frequent data swaps have low impact on off-package DRAM, whereas they will add overhead and wear out PCM memory quickly since PCM has higher write latency, higher write energy and limited write endurance. Thus, migrating the requested data on demand is not suitable for hybrid memory systems where the slow memory is a NVM. A lot of work regarding DRAM/NVM hybrid memory systems [7, 8, 9, 10] has been proposed to migrate hot pages to DRAM and store cold pages in NVM. PDRAM [7] tracks page access in the memory controller where each page is associated with a write counter. Hot pages whose write count are larger than a given threshold are migrated to the fast memory. However, the migrations target frequently written pages, leaving read-intensive pages in the slow memory. HSCC [10] tracks both read and write accesses via extending page table and TLB. Recording page access in the TLB is straightforward, but incurs high synchronization overhead due to page sharing between different cores, degrading system performance. RaPP [9], a page placement policy, adopts a modified multi-queue (MQ) [65] to rank memory pages. MQ defines M LRU queues of page descriptors, numbered from 0 to $M-1$. Each descriptor includes the physical page number (PPN), a reference counter, and an expiration time. The descriptors in queue $M-1$ represent the pages that are most frequently accessed. On the first access to a page, its descriptor is placed in the tail of queue 0. At the same time, its expiration time is set to $CurrentTime + LifeTime$. Every time the page is accessed, its reference counter is increased by 1. If a descriptor is in queue i, it will be upgraded to queue $i+1$ once its reference counter reaches 2^{i+1}. On the contrary, a page will be demoted and its descriptor is placed at the tail of the immediately inferior queue if the page is not accessed before it is expired. A page demoted twice without any intervening accesses is removed from the MQ. A page stored in the slow memory is scheduled for migration to the fast memory after its reference counter reaches the migration threshold. This approach is demonstrated to be effective, but it requires a sophisticated memory controller to maintain the MQ structure. It is costly to re-design the memory controller. In summary, the ways used to profile memory access in existing hybrid memory systems are costly due to redesigning the memory controller or extending TLB. Moreover, the hot page stays at the slow memory until its access count exceeds the migration threshold, which could miss lots of opportunities to improve performance. To solve this problem, we propose a cost-effective and energy-efficient architecture for die-stacked DRAM/NVM memory systems, especially for HBM/PCM memory systems, called Dual Role HBM (DR-HBM).

4.3 Architecture and Design

As PCM is slower than DRAM and has limited write endurance, researchers have proposed to employ a small DRAM buffer on top of PCM [68, 75] to shorten access latency and reduce the number of writes to PCM. However, this approach is not suitable for the hybrid memory system composed of HBM and PCM due to two reasons. First, HBM could be up to tens of gigabytes as the die-stacking technology becomes mature. Thus, using HBM as a cache would squander a non-negligible portion of memory space as HBM is not visible to OS. Second, architecting HBM as a huge cache between the LLC and main memory incurs high storage overhead for tags. For example, 1GB DRAM cache with 64-byte blocks requires 96MB of tag storage [39]. Based on these reasons, we divide HBM into two parts. A very small part (e.g. 128MB), called HBM cache, is used as a hardware-managed cache and the

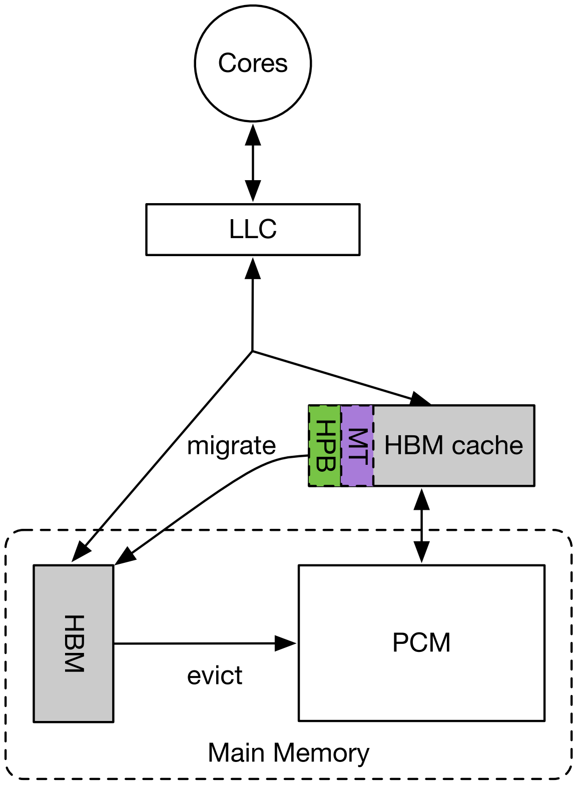


Fig. 20. The architecture of DR-HBM. The miss table (MT) and hot page buffer (HPB) are reserved in HBM cache.

remaining is used as a part of main memory, as shown in Figure 20. It is worth noting that the HBM cache is only used to cache the pages residing in the PCM and will be bypassed if the requested pages are located at HBM. Our design goal is to cache the hot pages in a short period of time at the HBM cache and store long-term hot pages at HBM directly. In doing so, most requests will be serviced either from the HBM cache or from HBM directly, minimizing the access to PCM. As a result, the performance is improved and the lifetime of PCM is prolonged.

4.3.1 HBM Cache Design

The HBM cache is designed to bridge the latency gap between LLC and PCM, and absorb write requests to prolong PCM's lifetime. In order to achieve this goal, we need to ensure a high hit ratio of the HBM cache and a low miss penalty. As most frequently accessed data lines have already been filtered by CPU caches (i.e. L1, L2, and LLC), the temporal locality is poor at the main memory layer [41]. We adopt a page granularity at which requested data is loaded to the HBM cache to exploit spatial locality. As a result, the hit ratio of HBM cache can be improved. When a request is missed at the HBM cache, the entire requested page is read out from the PCM memory and then is written to the HBM cache. The miss penalty is directly related with PCM's row buffer management and address mapping policies. In order to reduce the miss penalty, we use the open page [15], a row buffer management policy, for PCM to avoid activation operations for subsequent data lines in the requested page. Moreover, we adopt the "channel:row:column:bank:rank"address mapping policy to maximize the row hit rate of PCM as the row bits are placed as most significant bits (MSB). In doing so, the missed page can be loaded to the HBM cache quickly to reduce miss penalty.

4.3.2 HBM Cache Bypassing

Compared with conventional DRAM, PCM provides higher density but it has longer access latency. Hence, PCM has lower bandwidth than DRAM with the same number of channels and ranks. Due to the memory wall problem, lower bandwidth could make this problem even worse, leading to performance degradation. Therefore, we should alleviate PCM traffic as much as possible. We adopt page granularity to load pages to the HBM cache. On one hand, the coarse granularity can improve

Table 8. Workload Statistics.

Memory intensity	Workload	Percent of singleton pages	Hot page min # access	Hot page page percent	LLC MPKI	Memory footprint
Low	gobmk	15.9%	67	37.4%	3.0	848MB
	astar	6.6%	128	38.4%	3.4	4.6GB
Medium	facesim	16.2%	64	30.1%	9.3	7.5GB
	mcf	23.8%	343	24.6%	10.1	786MB
	sjeng	3.8%	128	17%	16.7	5.5GB
	libquantum	22.6%	262	56.4%	13.6	1.1GB
High	bwaves	14.9%	117	37.2%	27.1	23.8GB
	milc	14.2%	64	41.5%	23.2	10.9GB
	cactusADM	18.8%	56	11.5%	23.1	17.3GB
	canneal	17.6%	68	18.9%	26.6	5.4GB
	lbm	11.6%	128	44.2%	31.0	12.8GB

hit ratio of the HBM cache. On the other hand, the page granularity could waste bandwidth as some fetched pages could not be reused before they are evicted from the HBM cache. Our experiments show that a non-negligible portion of pages contains only single useful data line (typically 64 bytes). Such pages are called singleton pages [41]. Table 8 shows most workloads have more than 10% of singleton pages. As caching singleton pages wastes PCM bandwidth and HBM cache capacity, we devise a simple but effective policy, called cache on the second miss (CSM), to avoid caching singleton pages in the HBM cache. To achieve that, we maintain a miss table (MT) to record which pages are missed one time at the HBM cache. The MT and HBM cache are accessed in parallel, as shown in Figure 20. When a request missed at the LLC reaches the HBM cache, the request is serviced from the HBM cache if it is a hit. There are two cases if a request is missed at the HBM cache: 1) The requested PPN does not exist in the miss table. The requested page is treated as a singleton page. Therefore, the HBM cache is bypassed and the request is serviced directly from PCM. Meanwhile, the requested PPN is inserted into the miss table. In this case, only the requested data line is read out instead of reading the entire requested page from PCM to save bandwidth; 2) The requested PPN exists in the miss table. As the first missed data line is likely cached in CPU caches, the second miss indicates the requested page is not a singleton page. Hence, the entire requested page is loaded to the HBM cache and the requested PPN is removed from the miss table. In a word, CSM increases the effectiveness of HBM cache and reduces PCM traffic by avoiding to cache singleton pages, thereby improving system and saving energy.

4.3.3 Page Migration

In an HBM/PCM hybrid memory system, we face the same problem as other hybrid memory systems - which pages should be placed in the HBM. That is important for performance and PCM's lifetime. As tracking all page accesses is very costly, DR-HBM only tracks active pages that are cached in the HBM cache since tracking non-active pages is useless. Each cached page is tracked by using a counter to record the number of accesses. Compared with previous page tracking methods, our method is cost-effective due to two reasons. 1) The HBM cache is very small compared to the entire memory space; 2) DR-HBM does not need additional hardware support. When a page is evicted from the HBM cache and its access count is higher than a given threshold that is called migration threshold, the page is identified as a hot page. To figure out the migration threshold, we have conducted experiments on a test system where main memory only consists of PCM. The HBM cache with the same size as DR-HBM system is applied on top of the PCM to profile the minimum number of access of hot pages during their residency in the HBM cache. In our experiments, we define a workload's hot pages as the most frequently accessed pages that contribute to 70% of total page accesses. A hot page could be loaded to the HBM cache for many times, we only record the highest number of access among all residencies in the

HBM cache. Table 8 shows most workloads' hot pages are accessed at least 64 times during their residencies in the HBM cache. Therefore, we choose 64 as the migration threshold in our system.

The page migration can be performed either by hardware or by OS. The hardware-managed page migration is transparent to OS and can be done quickly. However, it needs complicated hardware support and also need to maintain a remapping table to record new page mappings. For each request missed in the LLC, the remapping table has to be looked up, adding extra latency for memory access. For the OS-managed page migration, the OS directly updates page table instead of maintaining a remapping table, and issue necessary TLB flushing instructions on each core of the system where there potentially is a stale TLB entry to keep consistent with the OS page table. This process is know as a TLB shoot-down. TLB shoot-down incurs high overhead and impacts system performance [78, 79]. As a result, the OS-managed page migration cannot occur frequently. Mark et al. [55] propose a hardware-assisted TLB shoot-down to speed up the TLB shoot-down process by using a specific hardware. However, we try to avoid using complicated hardware to make our design to be practical. Therefore, we adopt the OS-managed approach to migrate hot pages. To cope with the high overhead of TLB shoot-down, we propose a lazy migration policy by deferring page migrations. In other words, when a page is evicted from the HBM cache and its access count is higher than the migration threshold, the page is copied to a buffer, called hot page buffer (HPB), instead of migrating the hot page to HBM immediately. When the HPB is full, all buffered hot pages are migrated to HBM in a batch. After the migration process finishes, the page table is updated and TLBs are flushed in a batch to amortize the shoot-down overhead [80]. To support page table update, We maintain a global inverted page table to translate physical addresses to virtual addresses. Different virtual pages mapped to the same physical page are stored in a linked list. In order to service read requests for hot pages that are being migrated, the HPB is cleared after the new page mappings are finalized. Therefore, the HPB is still accessible during migration process and is designed to be accessed in parallel with the HBM cache to improve performance, as shown in Figure 20. However, the write requests and following read requests for being migrated pages must be paused until the migration finishes [81]. These requests have to be done at the new memory location for correctness. Long write pauses hurt application responsiveness and performance. Although a large HPB is helpful to amortize TLB shoot-down overhead, we adopt a HPB with 32 entries (i.e. 32 pages) as a good trade off since a large HPB could prolong write pauses. Moreover, we use two hot page buffers to store evicted hot pages from the HBM cache alternatively. When a HPB is full and is scheduled to be migrated, the other HPB is used to store evicted hot pages. In doing so, the migration and eviction processes can be proceeded in parallel.

4.3.4 Write Reduction

As the write requests to PCM not only wear it out, but also increase the effective read latency by almost 2X, causing significant performance degradation [64, 71], the writes to PCM should be minimized as much as possible. To this end, we analyze four possible cases that can occur when a page is evicted from the HBM cache, as shown in Table 9. Hot pages do not need to be written back to PCM whether they are dirty or not. However, the cold and dirty pages still need to be written back to PCM . Based on these observations, we reduce the number of writes from two aspects. First, as the hot pages are expected to be migrated to HBM, we devise a new page replacement policy, called Hot First LRU (HF-LRU). The HBM cache is implemented as a set associative cache. When a set is full and a page needs to be evicted, the oldest hot page is selected for cache replacement. If there is no hot page in this set, then the traditional LRU policy is applied to choose the least recently used page for replacement. Second, due to the asymmetric read/write performance of PCM, we increase the weight of write to migrate more write intensive pages to HBM. The access count is increased by two for each write request while it is increased by one for each read request. In doing so, write intensive pages get higher chance to become hot pages. reducing the number of write-back. In other words, the probability of occurrence of case 4 is reduced. As a result, the number of writes to PCM is reduced by applying these two methods together.

Table 9. Page Eviction Cases in HBM Cache.

Case	Hot	Dirty	Written back
1	✓	×	×
2	✓	✓	×
3	×	×	×
4	×	✓	✓

4.4 Evaluation

4.4.1 Evaluation Methodology

We implement DR-HBM with zsim [82] and DRAMSim2 [26] simulators. Zsim is a fast x86-64 and Pin-based [83] multi-core simulator. We add page table and TLB modules to support page migration. DRAMSim2 is a cycle accurate and detailed memory system simulator and is modified to support multiple memory instances. We use two instances of DRAMSim2 with different configurations to model HBM and PCM. The evaluated memory system consists of 4GB HBM and 32GB PCM and is managed at a page granularity (4KB). The parameters of HBM are set according to the DDR3 specification [22] except bus frequency and width. We double the bus frequency and width as HBM has lower access latency and higher bandwidth than

Table 10. System Parameters.

CPU	
Core	16 cores, 3.2GHz in-order
L1-D/L1-I cache	8-way, 64KB/64KB, 2 cycles
L2 cache	8-way, private 256KB, 8 cycles
L3	16-way, shared 16MB, 24 cycles
HBM (4GB)	
Bus frequency	1.6GHz (DDR 3.2GHz)
Channels/Ranks/Banks	8/1/8
Bus Width	128 bits per channel
tCAS-tRCD-tRP-tRAS	11-11-11-28 (cycles)
HBM cache	128MB, 16-way HF-LRU replacement
PCM (32GB)	
Bus frequency	400MHz (DDR 800MHz)
Channels/Ranks/Banks	2/1/8
Bus Width	64 bits per channel
tCAS-tRCD-tRP-tRAS	11-40-100-52 (cycles)
Read/write on row buffer hit	1.72 pJ/bit
Read and write on row buffer miss	79.46 pJ/bit and 1642.75 pJ/bit

DDR3. Timing and energy parameters of PCM are referred to [69]. We use a system composed of the same amount of HBM and PCM without page management as our baseline. We also implement CAMEO [48] and RaPP [9], two state-of-the-art hybrid memory systems, for comparison. Table 10 shows the system configuration in our study.

We evaluate a number of workloads with different memory access patterns from SPEC CPU2006 [84] and PARSEC 2.1 [27]. Gobmk, astar, mcf, sjeng, libquantum, bwaves, milc, cactusADM and lbm are selected from SPEC CPU2006. Facesim and canneal are selected from PARSEC. Based on the LLC miss per thousand instruction (MPKI), we classify all workloads into three categories: low ($MPKI < 5$), medium ($5 \leqslant MPKI < 20$), high ($MPKI \geqslant 20$). The LLC MPKI indicates the memory intensity. We select variety of workloads with different memory intensities for evaluation. Table

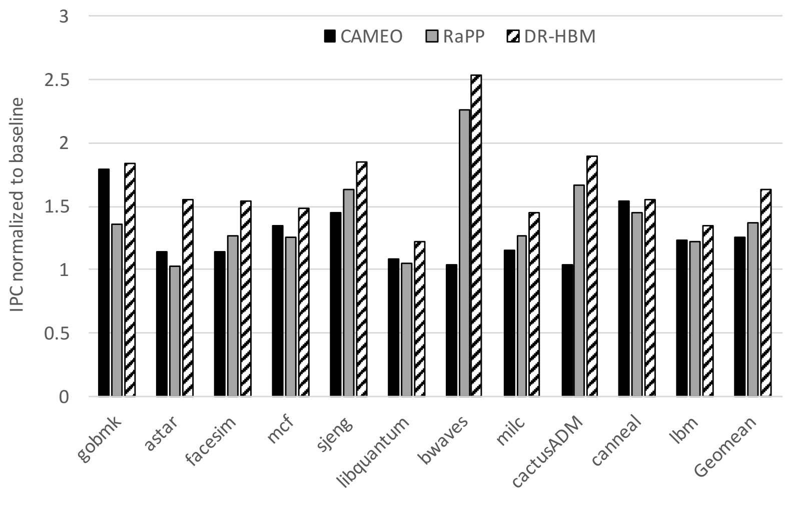


Fig. 21. Performance comparisons. On average, CAMEO and RaPP improve performance by 25% and 37.1%, respectively, as compared to the baseline, while DR-HBM improves performance by 63%.

8 shows the LLC MPKI and memory footprint for each workload. The evaluation is performed by launching 16 processes and each core executes a copy of the workload. In our experiments, the HBM cache is set to 128MB and the migration threshold is set to 64. We study the sensitivity of these two parameters in Section 4.4.6.

4.4.2 Performance Results

We compare the performance of DR-HBM with CAMEO and RaPP. We use instructions per cycle (IPC) as the performance metric. Figure 21 shows the performance results of these systems, which are normalized to the baseline. Compared to the baseline, CAMEO and RaPP improve the performance by 25% and 37.1% on average, respectively, while DR-HBM improves performance by 63% on average. Although RaPP outperforms CAMEO on average, CAMEO works better in some workloads, such as *gobmk*, *mcf*, *libquantum* etc. To understand the results, we also collect the percentage of requests serviced from the fast memory (i.e. the hit ratio

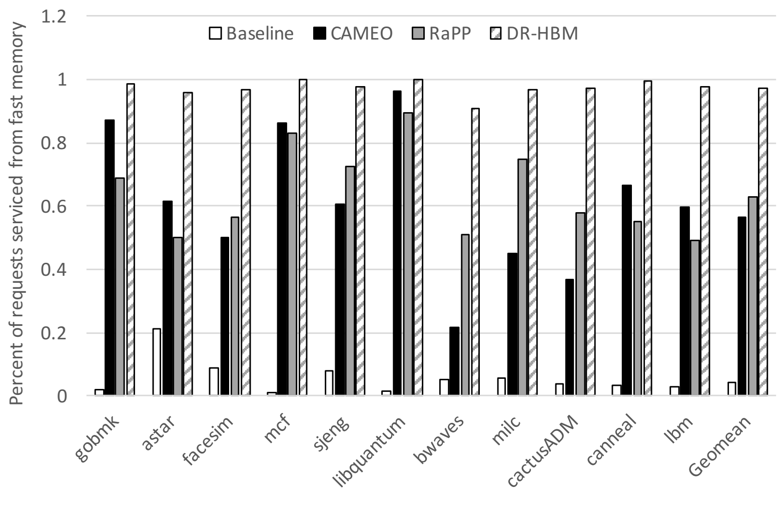


Fig. 22. Hit ratio of fast memory. Baseline, CAMEO, RaPP and DR-HBM achieve an average hit ratio of 4.2%, 56.6%, 63%, and 97.2%, respectively.

of fast memory), as shown in Figure 22. In our DR-HBM system, the hit ratio of fast memory is calculated as the number of requests that are serviced from HBM and the HBM cache divided by the number of total requests. From these two figures, it is clearly shown that the performance increases linearly with the hit ratio of fast memory. Without data migration, the baseline system only achieves an average hit ratio of 4.2%. However, CAMEO, RaPP and DR-HBM improve the hit ratio of fast memory to 56.6%, 63% and 97.2% on average, respectively, by migration hot data to the fast memory. As CAMEO swaps data at a fine (64B) granularity, RaPP should achieve a higher hit rate than CAMEO due to migrating data at a coarser (4KB) granularity. On the contrary, CAMEO achieves higher hit rate than RaPP in some workloads. As a result, CAMEO outperforms RaPP in these workloads. There are three reasons for this surprising results. First, these workloads have a good temporal locality as CAMEO could achieve a high hit ratio of fast memory; Second, a hot page in RaPP system has to wait until its reference count reaches the migration threshold before being migrated to the fast memory. As discussed in Section 4.2.2, the hit ratio of fast memory is reduce by deferring page migrations, while CAMEO swaps requested data on demand; Third, RaPP migrates data at a page granularity, which takes longer time than CAMEO to finish the migration. As a result, the hit ratio of fast memory is reduced further. Therefore, RaPP achieves lower hit ratio than CAMEO in the workloads where the temporal locality is strong. However, DR-HBM always achieves the highest hit ratio of fast memory among all evaluated systems due to three reasons. First, the requested pages are loaded to HBM cache on the second time, improving the hit ratio of HBM cache; Second, loading a page to fast memory is faster than swapping a page between slow and fast memories; Third, we alleviate the miss penalty of HBM cache by adopting open page policy and maximizing the row hit rate of PCM, as stated in Section 4.3.1. Consequently, DR-HBM outperforms CAMEO and RaPP constantly across all workloads.

4.4.3 Write Traffic on PCM

As the writes traffic on PCM not only wears it out, but also increases the effective read latency, write traffic on PCM is an important metric for evaluating our system. Figure 23 shows the write traffic on PCM of all evaluated memory systems. The results are normalized to the baseline. On average, CAMEO, RaPP and DR-HBM reduce the write traffic on PCM by 29.4%, 26.6% and 89.6%, respectively. CAMEO even increases the writes to PCM in some workloads, which is attributed to swapping requested data on demand. Although swapping the requested data to the fast memory (i.e. HBM) could potentially alleviate the write traffic on the slow memory (i.e. PCM), each swap also causes a write-back to the PCM. When the write traffic reduced by swapping cannot offset the write traffic increased by swapping, the write traffic on PCM is increased. For example, CAMEO generates 40% more write traffic on PCM than the baseline in workload *milc* as where CAMEO only gains 45% hit rate

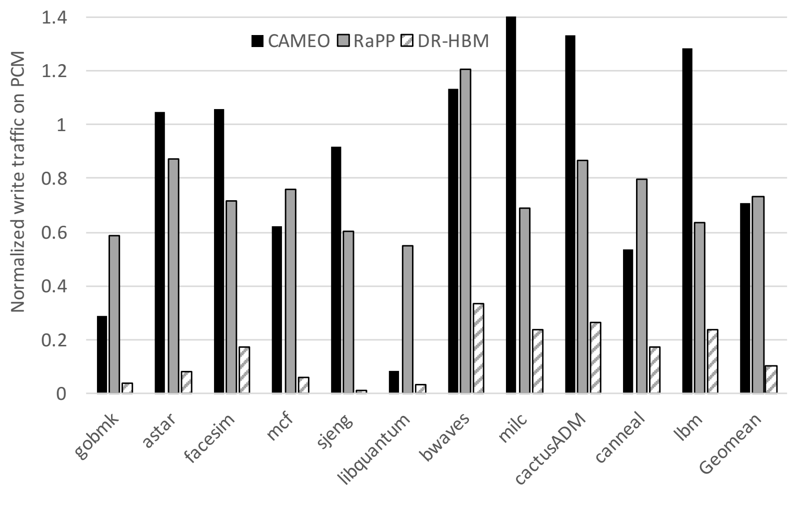


Fig. 23. The write traffic on PCM. Compared to the baseline, CAMEO, RaPP and DR-HBM reduce the write traffic on PCM by 29.4%, 26.6% and 89.6% on average, respectively.

of the fast memory. Similarly, RaPP also generates more writes to PCM than the baseline when the hit rate of fast memory is low, such as *cactusADM*. Since RaPP migrates hot data at a coarser granularity than CAMEO, RaPP generates more write traffic on PCM than CAMEO on average. However, DR-HBM reduces the writes to PCM significantly in all workloads due to two reasons. First, the HBM cache absorbs a lot of write requests, reducing the write traffic on PCM; Second, the proposed optimizations for write reduction reduce the number of write-back. As a result, the lifetime of PCM is prolonged. In our system, the main source of write traffic on PCM is the evictions from HBM. As the hot pages are migrated to HBM, the victim pages that are no longer hot are evicted and written back to PCM when the HBM is full. Hence, the workloads with more hot pages cause more write traffic on PCM. As shown in Table 8, the size of hot pages equals to the percent of hot pages times the whole workload's memory footprint. Workload *bwaves* that has the most hot pages causes the highest write traffic on PCM among all workloads.

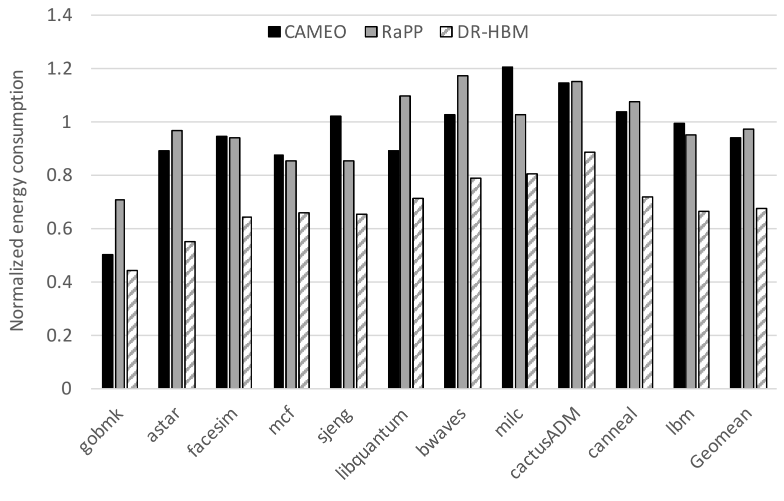


Fig. 24. Energy consumption. Compared to the baseline, CAMEO, RaPP and DR-HBM reduce the energy consumption by 6.3%, 3% and 32.9% on average, respectively.

4.4.4 Energy Consumption

To evaluate the energy efficiency of DR-HBM, we calculate energy consumption of all evaluated systems. We calculate the energy consumption of HBM based on Micron Power Calculator [61], while that of PCM is calculated based on the parameters shown in Table 10. Figure 24 shows the results, which are normalized to the baseline. On average, CAMEO, RaPP and DR-HBM reduce the energy consumption by 6.3%, 3% and 32.9%, respectively. Although CAMEO and RaPP improve the performance by 25% and 37.1%, respectively, CAMEO and RaPP only save little energy than the baseline due to additional energy consumed by data migration. The saved energy due to performance improvement is mostly offset by increased energy due to data migration. From Figure 23, both CAMEO and RaPP in workload *bwaves* generate more write traffic on PCM than the baseline. As a result, CAMEO and RaPP consume more energy than the baseline in *bwaves* workload. As shown in Table 10, PCM has asymmetric read and write energy cost. A write operation consumes several times higher energy than a read operation when they are both missed in the row buffer. Therefore, RaPP saves less energy than CAMEO as RaPP has higher write traffic on PCM on average. As DR-HBM always achieves the highest performance and the lowest write traffic on PCM across all workloads, DR-HBM is the most energy efficient system among all evaluated memory systems.

4.4.5 Overhead Analysis

In our experiments, we use 128MB out of 4GB HBM as a cache for the PCM. In the HBM cache, we reserve 3.5KB space for the miss table. There are 512 entries and each entry occupies seven bytes. And we also reserve space for two HPBs. As discussed in Section 4.3.3, each HPB has 32 entries. Each HPB entry consists of a PPN and 4KB data. Two HPBs consume 256.5KB HBM. In total, we reserve 260KB space in the HBM cache. We compare the storage overhead of DR-HBM with CAMEO [48] and RaPP [9]. Table 11 shows the storage overhead under a memory system composed of 4GB HBM and 32GB PCM. DR-HBM does not require any SRAM storage as DR-HBM does not need additional hardware support, which makes it easy to be implemented. However, RaPP needs a non-negligible space (126KB) in the memory controller. CAMEO consumes 2X HBM space as DR-HBM since CAMEO needs to store a remapping table. As DR-HBM allocates a small portion of HBM as a cache of PCM, DR-HBM consumes more HBM space than RaPP system, but it is still negligible (3.1%). However, the HBM cache improves the performance and reduces the number of writes to PCM. More importantly, the HBM cache is also used to identify hot pages. Therefore, DR-HBM is a cost-effective hybrid memory system.

Table 11. Storage Overhead Comparison

Storage	CAMEO	RaPP	DR-HBM
SRAM	512B	126KB	N/A
HBM	320MB (7.8%)	24MB (0.59%)	128MB (3.1%)

4.4.6 Sensitivity Study

4.4.6.1 Migration Threshold Sensitivity Analysis

We investigate the performance sensitivity of DR-HBM to the migration threshold. Figure 25 shows the performance under different migration thresholds. The results show there is no one migration threshold that fits all cases. For example, *gobmk*, *mcf* and *libquantum* workloads perform better with a lower migration threshold. As these workloads have a small memory footprint, lower migration threshold makes more pages be migrated to the fast memory, increasing the hit rate of fast memory. On the contrary, workloads with a large memory footprint, such as *bwaves* and *lbm*, gain better performance when the migration threshold is higher. The reason is that higher migration threshold can prevent over-migration, increasing the effective capacity of the fast memory and reducing the number of evictions from the fast memory to the slow memory. However, DR-HBM achieves the best performance on average when the migration threshold is set to 64. Therefore, we set the migration threshold to 64 in our experiments.

4.4.6.2 HBM Cache Size Sensitivity Analysis

We gauge the performance of DR-HBM while varying the size of HBM cache. Figure 26 shows the normalized IPC under different HBM cache sizes. Most workloads gain higher performance while increasing the HBM cache size. There is obvious performance improvement when the HBM cache is increased from 64MB to 128MB.

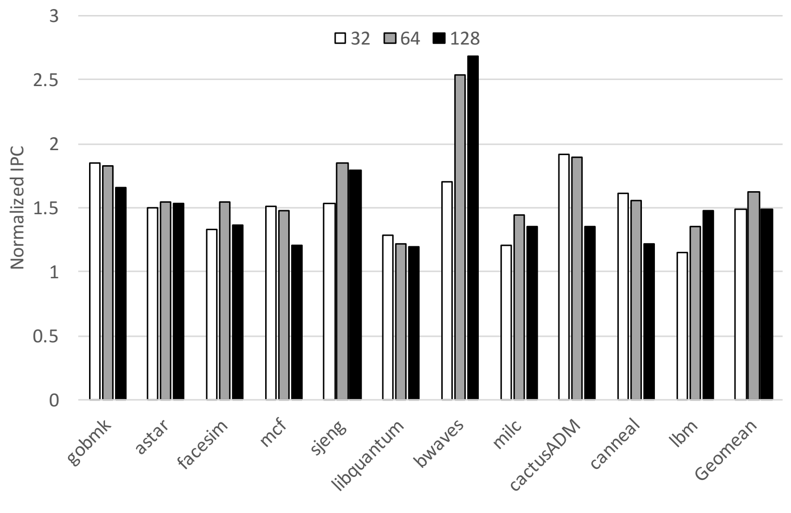


Fig. 25. Performance sensitivity to the migration threshold (32, 64, and 128).

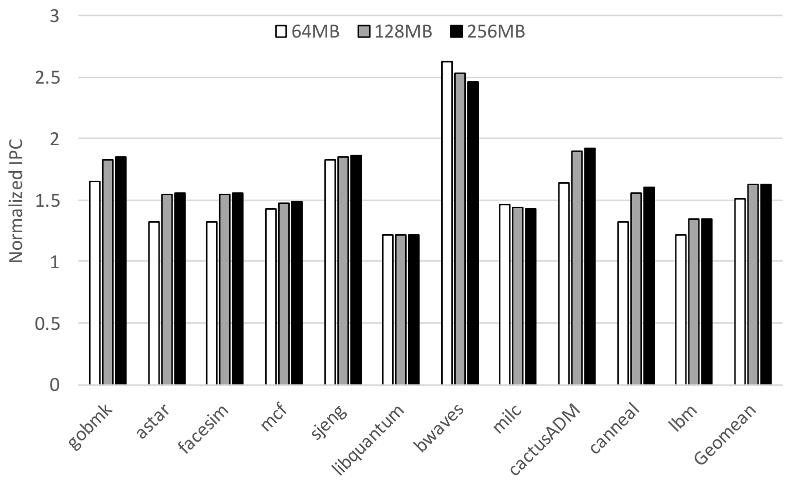


Fig. 26. Performance sensitivity to the HBM cache size (64MB, 128MB, and 256MB).

However, the performance of *bwaves* degrades while enlarging the HBM cache. In our design, the HBM cache is allocated from the HBM. The main memory space shrinks while enlarging the HBM cache. As *bwaves* has a large memory footprint, enlarging the HBM cache causes more page faults, leading to performance degradation. Since the performance can be improved marginally by increasing the HBM cache size after it reaches 128MB, we choose 128MB as a good HBM cache size.

4.5 Related Work

Hierarchical hybrid memory systems. A lot of previous work [68, 63, 39, 40, 41, 42, 43, 44, 51, 52, 53, 75, 85] has proposed architecting the fast memory as a cache/buffer to the slow memory. Qureshi et al. [68] propose using a small DRAM buffer on top of the PCM to improve the performance. Meanwhile, several techniques have been proposed to reduce the number of writes to the PCM memory and improve the wear-leveling. Jin et al. [75] propose a similar PCM-based hybrid memory system. In which, a small DRAM is used to cache writes to PCM pages. The DRAM buffer is managed by an age-based lazy caching policy (ALC). The ALC policy determines whether a PCM page is qualified to be cached in the buffer. A PCM page with higher write count has a higher chance to be buffered, reducing the writes to old pages. Therefore, the wear-leveling of PCM is improved. A lot of work regarding hybrid memory systems composed of die-stacked DRAM and off-package DRAM [63, 39, 40, 41, 42, 43, 44, 51, 52, 53] has been proposed. The main objectives of these work are to improve the hit ratio of DRAM cache, reduce the tag overhead and save off-package bandwidth. However, the main drawback of hierarchical hybrid memory systems is that the cache/buffer cannot contribute to the overall memory space. In this case, the system could lose a non-negligible portion of memory space when the fast memory becomes large. For example. HBM could be up to several gigabytes. Therefore, we architect HBM/PCM as a flat-addressable hybrid memory system.

Flat-addressable hybrid memory systems. Besides PDRAM [7] and RaPP [9] discussed in Section 4.2.2, prior work [10, 54] has been proposed to architect the fast memory as a part of main memory. The common idea is to migrate hot pages to the fast memory to improve system performance. To identify hot pages, these systems track page hotness by redesigning memory controller or extending TLB. Therefore, it is costly to implement these system. However, DR-HBM does not require any hardware changes. Islam et al. [74] demonstrate that prefetching is a effective technique for hybrid memory systems. A Markov-like prefetcher works better than CAMEO in some workload. Actually, the HBM cache of DR-HBM system is like a prefetcher and tracks page access of prefetched pages. In other words, we combine prefetching and profiling together in the HBM cache. Therefore, DR-HBM is a cost-effective architecture. Kannan et al. propose HeteroOS [76], which is an OS-level solution for managing memory heterogeneity in virtualized system. HeteroOS make the guest-OSes heterogeneity-aware and extracts rich OS-level information to provide smart memory placement reducing page migrations. Furthermore, HeteroOS combines the power of the guest-OSes information about applications with the hypervisor's hardware control to track page hotness and migrate hot pages to the fast memory. Compared with HeteroOS, DR-HBM is more generic as HeteroOS is designed for virtualized systems. Yu et al. [77] propose bandwidth-aware memory placement and migration policies for hybrid memory systems. These policies are orthogonal and can be applied in our work for page migration.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

In this dissertation, we make following contributions to improve the performance and energy efficiency of memory systems.

First, we propose *RPC* to alleviate DRAM refresh overhead. RPC allows concurrent refresh and memory access in a DRAM memory system by piggyback caching the to-be-read data to an adjacent rank as all ranks in the same channel are refreshed in a staggered fashion. As a result, read requests issued to a rank which is being refreshed can be serviced from the adjacent rank if the requested data is cached, without waiting for the refresh operation to complete. The implementation of *RPC* only requires minor modifications to the memory controller and negligible storage cost in each rank. Our evaluation results show that *RPC* outperforms *FGR* schemes and improves the system performance by 8.6% and 12.2% on average for PARSEC and SPLASH-2 benchmark suites, respectively.

Second, we propose SELF, a a high performance and bandwidth efficient approach to architecting on-chip DRAM as a part of memory. SELF selectively swaps lines in a requested page according to its page footprint instead of swapping an entire page blindly. In doing so, SELF increases the hit ratio of on-chip memory while avoiding swapping unnecessary lines to reduce off-chip bandwidth consumption. Moreover, SELF reuses the remapping page table to predict line location to reduce latency of off-chip accesses. As a result, SELF improves performance by 26.9% while reducing energy per access by 47.9% on average, compared to the baseline system of the same capacity.

Last, we propose DR-HBM, a cost-effective and energy-efficient architecture for hybrid HBM/PCM memory systems. In DR-HBM, the HBM plays two roles and is divided into two parts. A small portion of which, called HBM cache, is used as a cache for the PCM. The HBM cache is used to bridge the latency gap between LLC and PCM, and absorb write requests to prolong PCM's lifetime. Meanwhile, the HBM cache is also used to track page hotness without additional hardware support. The remaining HBM and PCM together constitute the main memory. Furthermore, we propose three techniques to improve performance and reduce writes to the PCM. First, CSM increases the effectiveness of HBM cache and reduces PCM traffic by avoiding to cache singleton pages which contain only single useful data blocks; Second, hot pages are migrated in batches to amortize TLB shoot-down overhead; Third, we propose Hot First LRU (HF-LRU) page replacement policy and increase the weight of write operations to reduce writes to the PCM. As we only exploit generic characteristics of HBM and PCM, DR-HBM is also applicable to other die-stacked DRAM/NVM memory systems. The experimental results show that DR-HBM outperforms two state-of-the-art hybrid memory systems, CAMEO and RaPP. Compared to the baseline without page management, DR-HBM improves the performance by 63% while reducing energy consumption by 32.9% on average.

5.2 Future Work

As there is no single memory technology that owns all desired features, such as high density, high bandwidth, and low power, hybrid/heterogeneous memory systems are very promising to meet the memory requirements of modern applications. In a heterogeneous memory system, there are multiple bandwidth sources with different bandwidths. As the memory wall problem continues to be a major performance bottleneck, how to fully exploit bandwidth that is available at the memory system is critical to the system performance, especially for bandwidth-intensive applications. DAP [85] points out delivered bandwidth of a hybrid memory system starts to decrease after the hit ratio of fast memory is higher than a certain number. Therefore, blind pursuit of high hit ratio of the fast memory will not result in high performance. Studies on how to dynamically control hot data migration between fast and slow memories to fully utilize all bandwidths in a hybrid memory system could be one of good directions for future research.

Moreover, in the DR-HBM, some workloads gain performance improvement while increasing the HBM cache size. However, other workloads' performance is degraded while increasing the HBM cache size, especially for the workload with a large memory footprint. Therefore, there no one cache size that can fit all cases, as discussed in Section 4.4.6.2. Studies on how to dynamically adjust the HBM cache size to achieve optimum performance according the workloads' access pattern could also be a good direction for the future work.

REFERENCES

[1] Jeffrey Stuechelix et al. “Elastic Refresh: Techniques to Mitigate Refresh Penalties in High Density Memory”. In: *Proceedings of the* 43rd *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2010.

[2] Young Hoon Son et al. “Reducing Memory Access Latency with Asymmetric DRAM Bank Organizations”. In: *Proceedings of the* 40th *Annual International Symposium on Computer Architecture (ISCA)*. 2013.

[3] Win. A. Wulf and Sally A. McKee. “Hitting the Memory Wall: Implications of the Obvious”. In: *ACM SIGARCH Computer Architecture News* 23.1 (Mar. 1995), pp. 20–24.

[4] Charles Lefurgy et al. “Energy Management for Commercial Servers”. In: *IEEE Computer* 36.12 (Dec. 2003), pp. 39–48.

[5] Kevin Lim et al. “Understanding and Designing New Server Architectures for Emerging Warehouse-Computing Environments”. In: *Proceedings of the* 35th *Annual International Symposium on Computer Architecture (ISCA)*. 2008.

[6] Mohammad Arjomand et al. “Boosting Access Parallelism to PCM-based Main Memory”. In: *Proceedings of the* 43rd *Annual International Symposium on Computer Architecture (ISCA)*. 2016.

[7] Gaurav Dhiman, Raid Ayoub, and Tajana Rosing. “PDRAM: A Hybrid PRAM and DRAM Main Memory System”. In: *Proceedings of the* 46th *ACM/IEEE Design Automation Conference (DAC)*. 2009.

[8] Wangyuan Zhang and Tao Li. “Exploring Phase Change Memory and 3D Die-Stacking for Power/Thermal Friendly, Fast and Durable Memory Architectures”. In: *Proceedings of the* 18th *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2009.

[9] Luiz Ramos, Eugene Gorbatov, and Ricardo Bianchini. “Page Placement in Hybrid Memory Systems”. In: *Proceedings of the* 25th *ACM International Conference on Supercomputing (ICS)*. 2011.

[10] Haikun Liu et al. “Hardware/Software Cooperative Caching for Hybrid DRAM/NVM Memory Architectures”. In: *Proceedings of the* 31st *ACM International Conference on Supercomputing (ICS)*. 2017.

[11] JEDEC. *DDR3 SDRAM Standard*. July 2010. URL: `http://www.jedec.org/standards-documents/results/jesd79-3f`.

[12] JEDEC. *DDR4 SDRAM Standard*. Sept. 2012. URL: `http://www.jedec.org/standards-documents/docs/jesd79-4`.

[13] Yuhua Guo et al. “Alleviating DRAM Refresh Overhead via Inter-rank Piggyback Caching”. In: *Proceedings of IEEE* 23rd *International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 2015.

[14] Yoongu Kim et al. “A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM”. In: *Proceedings of the* 39th *Annual International Symposium on Computer Architecture (ISCA)*. 2012.

[15] Dimitris Kaseridis, Jeffrey Stuecheli, and Lizy Kurian John. “Minimalist Open-page: A DRAM Page-mode Scheduling Policy for the Many-core Era”. In:

Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). 2011.

[16] Seongil O et al. “Row-Buffer Decoupling: A Case for Low-Latency DRAM Microarchitecture”. In: *Proceedings of the 41st Annual International Symposium on Computer Architecture (ISCA)*. 2014.

[17] Heesang Kim et al. “Characterization of the variable retention time in dynamic random access memory”. In: *IEEE Transactions on Electron Devices* 58.9 (2011), pp. 2952–2958.

[18] Prashant Nair, Chia-Chen Chou, and Moinuddin K. Qureshi. “A Case for Refresh Pausing in DRAM Memory Systems”. In: *Proceedings of the IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. 2013.

[19] Micron Technology. *1Gb: x4, x8, x16 DDR3 SDRAM*. 2006.

[20] Micron Technology. *2Gb: x4, x8, x16 DDR3 SDRAM*. 2006.

[21] Micron Technology. *4Gb: x4, x8, x16 DDR3 SDRAM*. 2009.

[22] Micron Technology. *8Gb: x4, x8 1.5V TwinDie DDR3 SDRAM*. 2011.

[23] Tao Zhang et al. “CREAM: a Concurrent-Refresh-Aware DRAM Memory Architecture”. In: *Proceedings of the IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 2014.

[24] Engin Ipek et al. “Self-optimizing memory controllers: A reinforcement learning approach”. In: *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA)*. 2008.

[25] Kevin Chang et al. “Improving DRAM Performance by Parallelizing Refreshes with Accesses”. In: *Proceedings of the* 20th *International Symposium on High-Performance Computer Architecture (HPCA)*. 2014.

[26] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. “DRAMSim2: A Cycle Accurate Memory System Simulator”. In: *IEEE Computer Architecture Letters* 10.1 (Jan. 2011), pp. 16–19.

[27] Christian Bienia and Kai Li. “PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors”. In: *Proceedings of the* 5th *Annual Workshop on Modeling, Benchmarking and Simulation*. 2009.

[28] Steven Cameron Woo et al. “The SPLASH-2 programs: Characterization and methodological considerations”. In: *Proceedings of the* 22nd *Annual International Symposium on Computer Architecture (ISCA)*. 1995.

[29] Scott Rixner et al. “Memory Access Scheduling”. In: *Proceedings of the* 27th *Annual International Symposium on Computer Architecture (ISCA)*. 2000.

[30] Avadh Patel et al. “MARSSx86: A Full System Simulator for x86 CPUs”. In: *Proceedings of the* 48th *ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2011.

[31] Mrinmoy Ghosh and Hsien-Hsin S. Lee. “Smart Refresh: An Enhanced Memory Controller Design for Reducing Energy in Conventional and 3D Die-Stacked DRAMs”. In: *Proceedings of the* 40th *ACM/IEEE International Symposium on Microarchitecture (MICRO)*. 2007.

[32] Jamie Liu et al. “RAIDR: Retention-Aware Intelligent DRAM Refresh”. In: *Proceedings of the* 39th *Annual International Symposium on Computer Architecture (ISCA)*. 2012.

[33] Janani Mukundan et al. “Understanding and Mitigating Refresh Overheads in High-Density DDR4 DRAM Systems”. In: *Proceedings of the* 40th *Annual International Symposium on Computer Architecture (ISCA)*. 2013.

[34] Jamie Liu et al. “An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms”. In: *Proceedings of the* 40th *Annual International Symposium on Computer Architecture (ISCA)*. 2013.

[35] Samira Khan et al. “The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study”. In: *Proceedings of the 2014 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. 2014.

[36] Yoongu Kim et al. “Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors”. In: *Proceedings of the* 41st *Annual International Symposium on Computer Architecture (ISCA)*. 2014.

[37] Ravi K. Venkatesan, Stephen Herr, and Eric Rotenberg. “Retention-Aware Placement in DRAM (RAPID): Software Methods for Quasi-Non-Volatile DRAM”. In: *Proceedings of the* 12th *International Symposium on High-Performance Computer Architecture (HPCA)*. 2006.

[38] Song Liu et al. “Flikker: Saving DRAM Refresh-power through Critical Data Partitioning”. In: *Proceedings of the* 16th *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2011.

[39] Gabriel H. Loh and Mark D. Hill. “Efficiently Enabling Conventional Block Sizes for Very Large Die-stacked DRAM Caches”. In: *Proceedings of the* 44th

Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). 2011.

[40] Moinuddin K. Qureshi and Gabriel H. Loh. "Fundamental Latency Trade-offs in Architecting DRAM Caches". In: *Proceedings of the* 45^{th} *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2012.

[41] Djordje Jevdjic, Stavros Volos, and Babak Falsafi. "Die-Stacked DRAM Caches for Servers Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache". In: *Proceedings of the* 40^{th} *Annual International Symposium on Computer Architecture (ISCA)*. 2013.

[42] Djordje Jevdjic et al. "Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache". In: *Proceedings of the* 47^{th} *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2014.

[43] Nagendra Gulur et al. "Bi-Modal DRAM Cache: Improving Hit Rate, Hit Latency and Bandwidth". In: *Proceedings of the* 47^{th} *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2014.

[44] Cheng-Chieh Huang and Vijay Nagarajan. "ATCache: Reducing DRAM cache Latency via a Small SRAM Tag Cache". In: *Proceedings of the* 23^{rd} *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2014.

[45] Micron Technology. *Hybrid Memory Cube*. URL: `https://www.micron.com/products/hybrid-memory-cube/short-reach-hmc/4GB#/`.

[46] JEDEC STANDARD. *High Bandwidth Memory (HBM) DRAM*. URL: `https://www.jedec.org/standards-documents/results/jesd235`.

[47] Jaewoong Sim et al. “Transparent Hardware Management of Stacked DRAM as Part of Memory”. In: *Proceedings of the* 47th *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2014.

[48] Chiachen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. “CAMEO:A Two-Level Memory Organization with Capacity of Main Memory and Flexibility of Hardware-Managed Cache”. In: *Proceedings of the* 47th *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2014.

[49] Yuhua Guo et al. “SELF: A High Performance and Bandwidth Efficient Approach to Exploiting Die-stacked DRAM as Part of Memory”. In: *Proceedings of IEEE* 25th *International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 2017.

[50] Berkin Akin, Franz Franchetti, and James C. Hoe. “Data Reorganization in Memory Using 3D-stacked DRAM”. In: *Proceedings of the* 42nd *Annual International Symposium on Computer Architecture (ISCA)*. 2015.

[51] Xiaowei Jiang et al. “CHOP: Adaptive Filter-Based DRAM Caching for CMP Server Platforms”. In: *Proceedings of the IEEE* 16th *International Symposium on High Performance Computer Architecture (HPCA)*. 2010.

[52] Sean Franey and Mikko Lipasti. “Tag Table”. In: *Proceedings of the* 21st *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2015.

[53] Yongjun Lee et al. “A Fully Associative, Tagless DRAM Cache”. In: *Proceedings of the* 42nd *Annual International Symposium on Computer Architecture (ISCA)*. 2015.

[54] Mitesh R. Meswani et al. “Heterogeneous Memory Architectures: A HW/SW Approach for Mixing Die-stacked and Off-package Memories”. In: *Proceedings of the* 21st *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2015.

[55] Mark Oskin and Gabriel H. Loh. “A Software-managed Approach to Die-stacked DRAM”. In: *Proceedings of the* 24th *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2015.

[56] Gabriel H. Loh et al. “Challenges in Heterogeneous Die-Stacked and Off-Chip Memory Systems”. In: *Proceedings of* 3rd *Workshop on SoCs, Heterogeneous Architectures and Workloads (SHAW)*. 2012.

[57] Ke Chen et al. “History-Assisted Adaptive-Granularity Caches (HAAG$) for High Performance 3D DRAM Architectures”. In: *Proceedings of the* 29th *International Conference on Supercomputing (ICS)*. 2015.

[58] Stephen Somogyi et al. “Spatial Memory Streaming”. In: *Proceedings of the* 33rd *Annual International Symposium on Computer Architecture (ISCA)*. 2006.

[59] Chi F. Chen et al. “Accurate and Complexity-Effective Spatial Pattern Prediction”. In: *Proceedings of the* 10th *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2004.

[60] Sanjeev Kumar and Christopher Wilkerson. “Exploiting Spatial Locality in Data Caches using Spatial Footprints”. In: *Proceedings of the* 25th *Annual International Symposium on Computer Architecture (ISCA)*. 1998.

[61] Micron Technology. *Calculating Memory System Power for DDR3*. Tech. rep. TN-41-01. 2007.

[62] Bharan Giridhar et al. "Exploring DRAM Organizations for Energy-Efficient and Resilient Exascale Memories". In: *Proceedings of the IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 2013.

[63] Xiangyu Dong et al. "Simple but Effective Heterogeneous Main Memory with On-Chip Memory Controller Support". In: *Proceedings of the IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 2010.

[64] Moinuddin K. Qureshi et al. "PreSET: Improving Performance of Phase Change Memories by Exploiting Asymmetry in Write Times". In: *Proceedings of the* 39th *Annual International Symposium on Computer Architecture (ISCA)*. 2012.

[65] Yuanyuan Zhou, James F. Philbin, and Kai Li. "The Multi-Queue Replacement Algorithm for Second-Level Buffer Caches". In: *Proceedings of USENIX Annual Technical Conference (ATC)*. 2001.

[66] Yuhua Guo et al. "A Cost-effective and Energy-efficient Architecture for Die-stacked DRAM/NVM Memory Systems". In: *under review (ICPP)*. 2018.

[67] J. Thomas Pawlowski. "Hybrid memory cube (HMC)". In: *Hot Chips Symposium (HCS)*. 2011.

[68] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. "Scalable High Performance Main Memory System Using Phase-Change Memory Technology". In: *Proceedings of the* 36th *Annual International Symposium on Computer Architecture (ISCA)*. 2009.

[69] Benjamin C. Lee et al. “Architecting Phase Change Memory as a Scalable DRAM Alternative”. In: *Proceedings of the* 36th *Annual International Symposium on Computer Architecture (ISCA)*. 2009.

[70] Ping Zhou et al. “A durable and energy efficient main memory using phase change memory technology”. In: *Proceedings of the* 36th *Annual International Symposium on Computer Architecture (ISCA)*. 2009.

[71] Moinuddin K. Qureshi, Michele M. Franceschini, and Luis A. Lastras-Montano. “Improving Read Performance of Phase Change Memories via Write Cancellation and Write Pausing”. In: *Proceedings of the* 16th *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2010.

[72] Mingzhe Zhang et al. “Balancing Performance and Lifetime of MLC PCM by Using a Region Retention Monitor”. In: *Proceedings of the* 23rd *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2017.

[73] Soyoon Lee, Hyokyung Bahn, and Sam H. Noh. “Characterizing Memory Write References for Efficient Management of Hybrid PCM and DRAM Memory”. In: *Proceedings of IEEE* 19th *International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 2011.

[74] Mahzabeen Islam et al. “Prefetching as a Potentially Effective Technique for Hybrid Memory Optimization”. In: *Proceedings of the* 2nd *International Symposium on Memory Systems (MEMSYS)*. 2016.

[75] Peiquan Jin et al. “A Page-Based Storage Framework for Phase Change Memory”. In: *Proceedings of the* 33rd *International Conference on Massive Storage Systems and Technology (MSST)*. 2017.

[76] Sudarsun Kannan et al. "HeteroOS - OS Design for Heterogeneous Memory Management in Datacenter". In: *Proceedings of the* 44th *International Symposium on Computer Architecture (ISCA)*. 2017.

[77] Seongdae Yu, Seongbeom Park, and Woongki Baek. "Design and Implementation of Bandwidth-Aware Memory Placement and Migration Policies for Heterogeneous Memory Systems". In: *Proceedings of the* 31st *ACM International Conference on Supercomputing (ICS)*. 2017.

[78] Carlos Villavieja et al. "DiDi: Mitigating the Performance Impact of TLB Shootdowns Using a Shared TLB Directory". In: *Proceedings of the* 20th *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2011.

[79] Daniel Lustig, Abhishek Bhattacharjee, and Margaret Martonosi. "TLB Improvements for Chip Multiprocessors: Inter-Core Cooperative Prefetchers and Shared Last-Level TLBs". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 10.1 (Apr. 2013), 2:1–2:38.

[80] Nadav Amit. "Optimizing the TLB Shootdown Algorithm with Page Access Tracking". In: *Proceedings of USENIX Annual Technical Conference (ATC)*. 2017.

[81] Santiago Bock et al. "Concurrent Page Migration for Mobile Systems with OS-Managed Hybrid Memory". In: *Proceedings of the* 11th *ACM Conference on Computing Frontiers (CF)*. 2014.

[82] Daniel Sanchez and Christos Kozyrakis. "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems". In: *Proceedings of the* 40th *International Symposium on Computer Architecture (ISCA)*. 2013.

[83] S. Naftaly. *Pin - A Dynamic Binary Instrumentation Tool.* https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[84] *SPEC CPU 2006.* http://www.spec.org/cpu2006.

[85] Jayesh Gaur et al. “Near-Optimal Access Partitioning for Memory Hierarchies with Multiple Heterogeneous Bandwidth Sources”. In: *Proceedings of the* 23^{rd} *IEEE International Symposium on High Performance Computer Architecture (HPCA).* 2017.

VITA

Yuhua Guo was born on January 3, 1985, in Ji'an City, Jiangxi Province, China. He graduated from Suichuan Middle School, Ji'an, China in 2003. He received his Bachelor of Science in Computer Science and Technology from Shaanxi University of Science and Technology, Xi'an, China in 2007. And he received his Master of Science in Computer Architecture from Huazhong University of Science and Technology, Wuhan, China in 2011.