The Pennsylvania State University
The Graduate School
College of Engineering

HARDWARE SOFTWARE CO-DESIGN FOR OPTIMIZING MEMORY HIERARCHY IN MANY-CORE AND MULTI-SOCKET SYSTEMS

A Dissertation in
Computer Science and Engineering
by
Jagadish B. Kotra

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2017

The dissertation of Jagadish B. Kotra was reviewed and approved* by the following:

Mahmut Taylan Kandemir
Dissertation Advisor, Chair of Committee
Professor, Dept. of Computer Science and Engineering

Mary Jane Irwin
Emeritus Evan Pugh Professor and A. Robert Nall Chair
Dept. of Computer Science and Engineering

Kamesh Madduri
Assistant Professor
Dept. of Computer Science and Engineering

Dinghao Wu
Associate Professor
College of Information Sciences and Technology

Chita R. Das
Distinguished Professor
Head of the Dept., Computer Science and Engineering

*Signatures are on file in the Graduate School.

Abstract

Thanks to Moore's law, the number of transistors on a chip have been increasing over time without increasing area of the processing die. The increased number of transistors are being invested in separate cores instead of optimizing the already complex out-of-order cores to ensure the power-density ie., the heat dissipated per unit area is not too high. Hence, the complex uni-core systems have paved way in to multi- and many-core systems on a processor die of necessarily the same size, thereby resulting in increased amount of processing per unit area. Similar to the processing-end, the number of transistors on the memory side have also increased (though not at the same rate), resulting in the increased memory (DRAM) capacity over the years.

Such increased number of transistors at the processor- and memory-ends have enabled significant computation and memory capacity scalings over time in the same area. However, the speed-ups observed due to the increased processing power were not linear. This was because the number of pins that connect the processor and memory haven't been increased as that would make the die size bigger. As a result, with the increased number of cores, the effective memory bandwidth per computation core decreased over time. Apart from the reduced memory bandwidth per core, the increased memory density (capacity per unit area) resulted in interesting performance and power ramifications in DRAM. Due to the volatile nature of the DRAM, the increased memory density warranted more number of rows to be refreshed in effectively the same retention time. As a result, certain sections of DRAM remained inaccessible to continuously feed the data in to the processing elements resulting in reduced overall memory bandwidth as well.

As a result, the performance-gap between the processor and memory have increased significantly over time. This gap in performance between processor and memory is widely referred to by the researchers as "memory-wall".

In my thesis, I have proposed various techniques to bridge the performance-gap

between the processor and memory. The techniques I have proposed can be broadly be classified in to:
1. Entirely hardware-based proposals,
2. Entirely software-based proposals, and
3. Hardware-Software based co-design proposals.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I had the most memorable time during my graduate life at Penn State. For that, so many people have contributed in many different ways. First amongst all of them is my advisor Dr. Mahmut Taylan Kandemir.

Dr. Kandemir gave me all the freedom in pursuing the research problems of my interest. He was always helpful and professionalistic in his approach. I still remember how he woke up in the middle of a night during my first paper submission so that I could have the feedback timely for my early morning submission. I am extremely thankful for that. Such instances speak volumes about his professionalism and his kindness towards helping students. Dr. Kandemir inspired me in many ways on how to lead a true researcher life and I am extremely fortunate to have him as my academic advisor.

I would also like to take this opportunity to thank my other committee members: Dr. Mary Jane Irwin, Dr. Kamesh Madduri and Dr. Dinghao Wu. They made a positive impact in shaping my overall dissertation. I fondly remember how I was introduced to the field of Computer Architecture when I took the course with Dr. Mary Jane Irwin in my first semester. Dr. Kamesh Madduri provided interesting insights in one of the projects we collaborated. Dr. Dinghao Wu provided me with timely feedback which helped me in making this dissertation more solid.

My research could not have taken shape without my labmates and collaborators: Praveen Yedlapalli, Chun-yi Liu, Anup Sarma, Mohammad Arjomand, Haibo Zhang, Prashanth Thinakaran, Jashwanth Raj, Diana Guttman, Narges Shahidi, Karthik Swaminathan, Wonil Choi, Ashutosh Pattnaik, Nachiappan Chidambaram, Jihyun Ryoo et al. I thank them for their help and friendship. Their patient ears gave me the solace amidst paper rejections. Apart from research, my squash/badminton buddies: Mike Bluementhal (late), Shakil et al helped me remain sane amidst long working hours.

I would also like to thank Alaa R. Alameldeen, Chris Wilkerson and Zeshan A. Chisthi from Intel Labs for providing me an internship opportunity. Our collaboration/discussions are fruitful and they helped me in advancing my research tremendously. Also, Seongbeom Kim for mentoring me during my internship stint at VMware. Similarly, Kathy Debnath and Adrian Moga for my first internship stint at Intel.

I would also take this opportunity to thank the CSE support staff and CSE helpdesk at Penn State for their invaluable service to CSE students. Especially, CSE helpdesk staff including Mark Tamminga, Kem Hurtley, Barbara Einfalt, Eric Prescott et al, have been very supportive in addressing my countless helpdesk requests during odd times (including nights and weekends). Needless to mention their invaluable contribution for this dissertation as they were instrumental in keeping the machines up and running.

I would like to express my deepest gratitude for my family, without whose support this dissertation would not have been possible. My mom, Kusuma Kumari Kotra, and dad, Sai Babu Kotra, always believed in me and encouraged me to pursue my dreams without any hesitation. I am indebted to my parents forever for their unconditional love and support. My sister, Jyostna Kotra, inspired me a lot in my life in general. I have learnt a lot from her, especially her patience and tenacity.

Finally, I thank my wife, Mythri Sunkara, who was patient enough to bear me amidst my hectic work schedules, ups and downs. She always ensured that I am at my best by taking good care of me. My Ph.D journey over the years would not have been the same without her constant support and motivation. I am very fortunate to have Mythri in my life.

I would also like to take one last opportunity to thank the National Science Foundation (NSF), Department of Energy (DOE), Intel and Microsoft for funding my research. This dissertation would not have been possible without their financial support.

Dedication

****** To my family and friends ******

Chapter 1

Introduction

With the increase in number of transistors following Moore's law, systems have transformed from single core to multicore to manycore processors. While the core frequency has saturated owing to the power-wall, increased number of transistors resulted in increased number of cores on-chip. Currently Intel Xeon-Phi has around 60 cores integrated on a chip, while each core itself is 4-way SMT making a total of 240 hardware threads. With the memory bandwidth doubling only every 4 years (on average), memory bandwidth fails to keep up with the processor throughput, there by increasing the difference between the speed of processors and memory. This performance gap between processor and memory is widely referred to as "memory wall" [1] by industry and academia.

To overcome this performance-gap between processor and memory, researchers employed solutions broadly falling in to following categories:

Dense last-level caches (LLC): Academia and industry have proposed employing wide variety of emerging memories including non-volatile STT-RAM, Re-RAM and volatile embedded(e)-DRAM [2], stacked 3D DRAMs [3, 4] as last-level caches. Employing such dense last-level caches alleviates the pressure on off-chip memory bandwidth.

Memory performance scaling: As the number of transistors enable scaling the memory densities rapidly, the non-proportional growth in memory bandwidth have resulted in memory refresh related overheads being the performance bottleneck. To alleviate these bottlenecks for the futuristic memories, researchers in academia and industry have introduced varying solutions ranging from stalling the refreshes to parallelizing the refreshes with accesses.

Processing-In-Memory solutions: These solutions leveraged the isolated peripheral layer in stacked 3D DRAMs to offload certain computations to the stacked DRAM. Such solutions averted the data-movement costs associated with offloaded computations thereby increasing the overall performance.

While these efforts from academia and industry demonstrate the importance of addressing the “memory wall”, there is still a lot of scope for unravelling the full-potential of these high-level solutions. The advent of these emerging and scalable memory solutions present interesting software and hardware challenges to computer architects to seemlessly integrate them in to the processor memory hierarchy. In my thesis, I try to explore various hardware-only, software-only and hardware-software co-design based solutions to bridge this performance-gap further.

My thesis is organized as follows:

In Chapter 2, I present the background information on the organization of manycore and large multi-socket processors. After the background on processor organization, I briefly cover the organization of memory hierarchy inside these processors. Specifically, I present a brief primer on DRAM organization, explaining the various performance bottlenecks as it scales further. After covering the basics on DRAM organization, I present details on how systems software manages the memory interms of allocation and unallocation. In Chapter 3, I present a hardware-only, Resistive-NUCA proposal, which addresses the wearout issue of Re-RAM based last-level caches in a performance-conscious manner. In Chapter 4, I present my software-only approach for detecting and addressing congestion in a multi-socket processor. In Chapter 5 I present my proposal on evaluating the potential benefits of on-chip near-data computing in manycore systems. While in Chapters 6 and 7, I present a hardware-software co-design based solutions to address the memory-wall.

Chapter 2

Background

2.1 Manycore and Multi-socket processors

In this sub-chapter, overview of manycore processor is presented showing various how data is accessed from the last-level L3 cache bank after a miss in the private L1 and L2 caches along with how an on-demand memory request is routed to the corresponding memory channel. Also, a brief overview on large NUMA systems is presented. Later in this chapter prior works related to this proposal is presented.

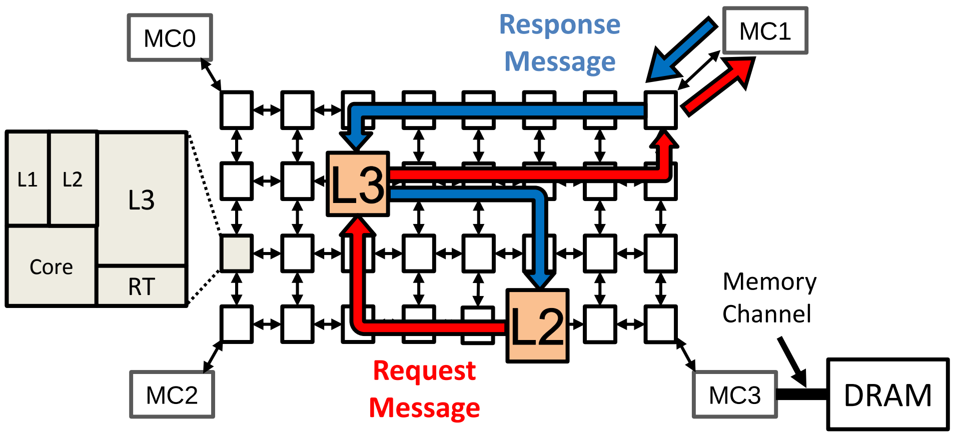
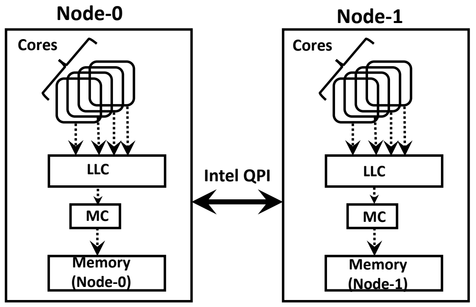
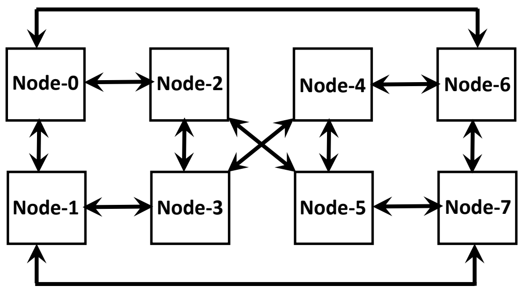


Figure 2.1: NoC-based manycore processor.



(a) Intel Haswell block diagram.



(b) Intel Westmere block diagram.

Figure 2.2: Multi-socket NUMA-based systems.

2.1.1 Overview of manycore processor

Figure 2.1 shows a NoC based many core processor with 32 cores. Each tile shown as shown in the figure consists of a core, a private L1/LD caches, a private L2 cache and a L3 cache bank. Also, a tile consists of a NoC router which routes the packets from one tile to another. The manycore shown in figure 2.1 contains 4 memory controllers (MC) at the four corners of a chip. Each memory controller manages the off-chip DRAM connected to the controllers over a memory channel. Traditional interconnects employ static X-Y routing where a packet is routed from source to the destination first in the X-direction and then in the Y-direction. Hence this X-Y routing is a deterministic routing.

The last level L3 cache banks can be managed either in the private cache configuration or in the shared cache configuration. In a private cache configuration, a core in a tile will only use the local cache bank and hence does not incur remote cache bank accesses. However, private cache configuration can cause under-utilization of the cache space but has the benefit of less interference. The other end of the spectrum of a shared cache configuration can result in better utilization of the cache banks and hence can increase the overall performance of the system.

In a shared configuration, the access to a cache line in local cache bank will incur lower access latency compared to the access to a cache line in the remote cache bank. Hence, the shared cache configuration will result in Non-Uniform Cache Access latencies, widely referred to as NUCA architecture. There are various cache line mapping schemes proposed by the researchers [5] [6] [7]. Prominent of these various NUCA configurations is the static-NUCA configuration where each cache line is mapped to a fixed LLC bank based on some bits in the physical address.

Figure 2.1 shows how a on-demand request is routed to a LLC L3 bank upon a miss in the private L2 bank. On-demand request first travels in the X-direction and then in the Y-direction to the destination L3 bank. If it results in a L3 hit, the cache block is returned to the core, else the on-demand request results in a memory access. Hence the request is routed to the corresponding memory controller based on the X-Y routing again as shown in figure 2.1.

From/To Cycles	N-0	N-1	N-2	N-3	N-4	N-5	N-6	N-7
N-0	**290**	454	735	736	840	835	839	843
N-1	454	**290**	734	748	865	860	864	868
N-2	735	734	**290**	452	839	863	888	863
N-3	736	748	452	**290**	840	861	862	864
N-4	839	862	839	840	**290**	451	734	750
N-5	835	860	863	861	451	**290**	741	739
N-6	839	863	885	863	734	729	**290**	454
N-7	843	868	863	864	748	739	454	**290**

Table 2.1: Intel Westmere bootup latencies.

From/To Cycles	N-0	N-1
N-0	**229**	319
N-1	319	**229**

Table 2.2: Intel Haswell bootup latencies.

2.1.2 Overview of a NUMA system

Figures 2.2a and 2.2b shows the block diagram of a Intel Haswell 2 socket (named as Node-0 and Node-1) and Intel Westmere NUMA systems. *Socket* and *Node* are used interchangeably in the rest of the proposal. All the sockets are connected through an Intel Quick Path Interconnect (QPI) [8] [9] [10]. A socket consists of a local memory which is managed by a local memory controller, represented by MC in the Figure 2.2a, and is a Chip Multi-Processor (CMP) containing cores; all cores share a last-level cache, represented by LLC in the figure. Upon a miss in the LLC, depending on where the data is allocated, data is fetched from either the local memory or the remote memory in a different socket.

A local memory access incurs a DRAM access delay and, if the MC is congested, a MC queuing delay as well. However, since, the remote memory access involves moving the data from the remote socket over the QPI, an additional interconnect latency is incurred apart from the MC queuing and DRAM access delays. Therefore, in a NUMA based system, local memory access incurs lower latency than the remote memory access. Tables 2.1 and 2.2 show the latencies in CPU cycles after a system bootup without any guest VMs running inside the ESXi hypervisor for Westmere and Haswell systems, respectively. In these tables, the value in row-x and column-y represent memory access latency in CPU cycles observed from Node-x when accessing the data allocated in the local memory of Node-y. Hence, values in the diagonal (bolded) represent the local memory access latencies. From these tables, it can be observed that the local latency incurred is always lower than the remote latency, and in the case of Westmere, remote latency increases with the distance to data.

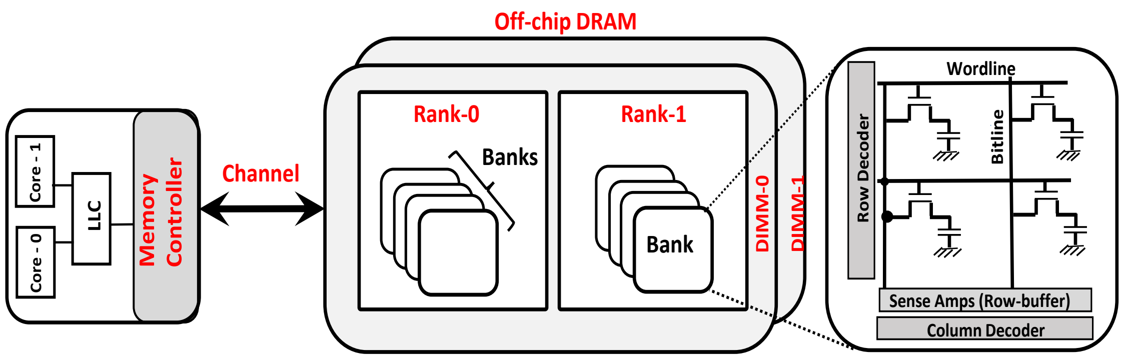
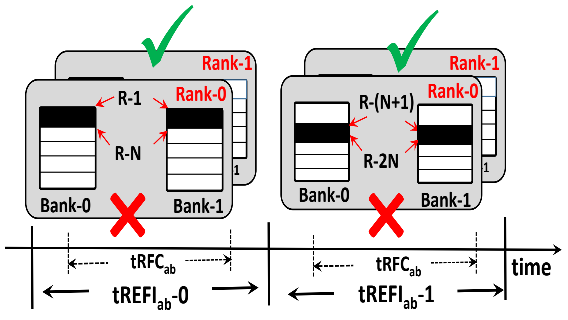


Figure 2.3: Basic DRAM organization.

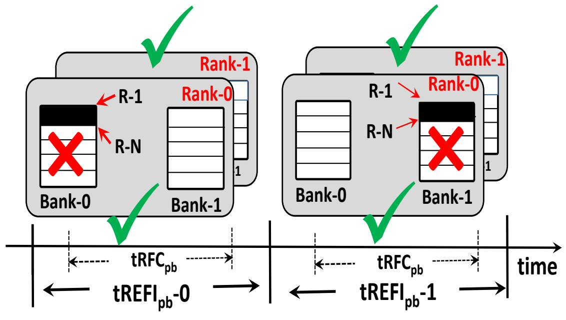
2.1.3 Off-chip DRAM Overview

2.1.3.1 DRAM Organization

As shown in Figure 2.3, a typical DRAM hierarchy is made up of channels, ranks and banks. Each on-chip memory controller (MC) manages corresponding DIMMs by issuing various commands over the command bus, and the corresponding data is traversed over the data bus. Each DIMM, as shown in Figure 2.3, is made up of multiple DRAM ranks, while each rank further consists of multiple banks, as also shown in the figure. Each bank consists of DRAM cells laid out in rows (typically the size of a DRAM page, 4KB or 8KB) and columns connected by wordlines and bitlines, respectively. The data in each DRAM row is accessed by activating the row into DRAM sense-amplifiers (also referred to as row-buffers) through a RAS command, after which a corresponding cache line from an activated row is accessed by a CAS command. Hence, a row-buffer caches the most recently opened row till it is precharged explicitly. Since accessing data from an already opened row (present in row-buffer) is faster, different row-buffer management policies have been proposed by researchers in the past [11] [12] [13], which aim at decreasing the overall memory latency.



(a)



(b)

Figure 2.4: (a) All-bank refresh. (b) Per-bank refresh with $tREFI_{pb} = tREFI_{ab}/(\text{numBanks})$.

2.1.3.2 DRAM Refresh Scheduling

DRAM cells are typically made up of an access transistor and a capacitor. Over time, DRAM cells leak charge and hence need to be refreshed periodically to maintain the data integrity. DRAM cell retention times (denoted by tREFW) are often a function of the operating temperatures and process variation [14] [15]. Typically, tREFW is 64msec for environments operating in temperatures $<$ 85 deg C, while it is halved to 32msec when temperature is beyond 85 deg C. Instead of refreshing all the DRAM rows at once, MC issues refresh command once in every refresh interval (denoted by tREFI). Typically, tREFI is in the order of μseconds and is generally 7.8 μsecs for DDR3, while finer refresh granularities are supported for DDR4 in the 2x and 4x modes, where tREFI is 3.9 μsecs and 1.95 μsecs, respectively [16]. Each refresh operation issued by an MC lasts for a refresh cycle time (denoted by tRFC), which is typically in the order of several *nano seconds*. tRFC is a function of the employed tREFI and the number of rows to be refreshed. tRFC increases with the increase in the density of DRAM [15] [16] [17], causing significant performance bottlenecks for high capacity DRAMs. Commercial DDR cells are refreshed at a rank level, while the mobile LPDDRs can be refreshed at a per-bank granularity. Figures 2.4a and 2.4b illustrate the refresh operations when rows are refreshed at a rank-level and bank-level, respectively, in a system comprising of 2 ranks and 2 banks per rank.

All-bank refresh: As shown in Figure 2.4a, a refresh operation issued at the rank-level refreshes a certain number of rows (say N) in all the banks in that rank. Figure 2.4a depicts that rows R-1 to R-N are refreshed in banks B-0 and B-1 during the first refresh interval (indicated by $tREFI_{ab}$-0), while rows R-(N+1) to R-2N are refreshed during the second refresh interval $tREFI_{ab}$-1. As can be observed in Figure 2.4a, in a given $tREFI_{ab}$, since all the banks in a rank are being refreshed, the entire rank-0 is not available as indicated by $\times$ in Figure 2.4a for rank-0 for $tRFC_{ab}$ duration, while rank-1 is available as indicated by $\checkmark$. Since the entire rank is not available to serve the on-demand memory requests during $tRFC_{ab}$, performance degradation is significant in all-bank refresh as opposed to per-bank refresh.

Per-bank refresh: To increase the availability of the number of banks during refresh, LPDDRs allow refresh commands to be issued at a bank granularity. Figure 2.4b depicts the per-bank refresh employed by LPDDRs. Since refreshes are issued at a bank granularity, the refresh interval employed by per-bank (denoted by $tREFI_{pb}$) is smaller than that of $tREFI_{ab}$ and $tREFI_{pb} = tREFI_{ab}$ / (numBanks). As can be observed in Figure 2.4b, rows R-1 to R-N in Bank-0 are refreshed in $tREFI_{pb}$-0; as a result, only Bank-0 is not available during $tRFC_{pb}$ (denoted by $\times$), while the other banks in Rank-0 are available to serve on-demand requests. In $tREFI_{pb}$-1, as can be observed from Figure 2.4b, the same rows R-1 to R-N in Bank-1 are refreshed, while Bank-0 is available for on-demand requests. Hence, banks in all the ranks are refreshed in a round-robin fashion in per-bank refresh [15]. Since not all the banks in a rank are refreshed in a given $tREFI_{pb}$ in per-bank refresh, performance degradation in per-bank refresh is not as catastrophic as in all-bank refresh.

2.2 Overview of System Software (OS)

2.2.1 Linux Memory Allocator

Linux uses a buddy memory allocator [18] to allocate physical addresses for applications. It maintains free-lists per zone to cater to the memory allocation requests. The traditional Linux memory allocator is oblivious to the DRAM bank organization, and consequently any given application can have memory allocated in all the DRAM banks depending on the memory footprint of the application. Such DRAM-oblivious memory allocation accommodates for better bank-level parallelism (BLP) for applications; however, in some multi-programmed environments, it can lead to memory interference as well [19] [20]. Such interference in the multi-programmed environments result in not just the contention for memory bandwidth but also poor row-buffer locality in DRAMs, thereby degrading performance. To avert this memory interference, researchers have proposed DRAM bank-aware memory partitioning [19] [20], where the OS memory allocator is aware of the hardware address-mapping, viz, channel, rank and bank bits and can allocate memory such that certain applications will access certain DRAM banks, reducing the interference. However, since such a partitioning limits the bank-level parallelism (BLP), researchers have also proposed dynamic mechanisms to balance BLP vs row-buffer locality [21]. Hence the OS memory allocator plays a crucial role in managing various shared on-chip resources including the memory bandwidth.

2.2.2 Linux Process Scheduling

Linux kernel past 2.6.23 version uses Completely Fair Scheduler (CFS) to schedule tasks across processor cores [22] [23]. CFS uses a notion called the "virtual runtime", which indicates the next time-slice[1] when a task[2] will be scheduled. CFS employs time-ordered red-black tree data structure where the tasks are sorted by the vruntime. The left-most task in the red-black tree is chosen by the CFS scheduler as it is the oldest executed task among the runnable tasks. CFS manages red-black tree per CPU and in a multi-CPU system, CFS runs the load-balancer in the background to maintain an equal number of tasks in the per-CPU queues to

[1]We use time-slice and time quantum interchangeably in this work.
[2]We use "application", "benchmark", and "task" interchangeably in this work.

maximize the overall throughput [23]. The time-slice of the CFS scheduler in Linux is typically in the order of 1-5msec [24][3]. Since the OS scheduler schedules the tasks on the CPU, it provides ample opportunities to schedule suitable tasks if some of the underlying hardware bottlenecks are exposed to the OS to improve the overall system throughput. In current-day systems, task scheduling in the OS is agnostic of the refresh scheduling in the DRAM. Such an independent schedule of tasks and DRAM refreshes causes significant performance problems. In this work, we propose hardware-software co-design where the OS partitions the memory across tasks thereby enabling the OS task scheduler to schedule processes in a *refresh-aware fashion*.

[3]We observed similar values for time-slice in our full-system experiments.

Chapter 3

Resistive-NUCA (Re-NUCA): A Hardware Approach

In this chapter, a pure hardware approach which wear-levels the Re-RAM cache banks in a performance-aware fashion is presented.

3.1 Introduction

Workloads in the next generation of large-scale computing systems are expected to be highly data-intensive and have large working-sets. The processing power is also steadily increasing and major manufacturers are planning to integrate hundreds of cores on a die. To mitigate performance loss due to increasing memory access rate in multi-core systems running multiple workloads, computer architects tend to employ high-capacity on-chip cache hierarchies. Nevertheless, performance is not the only efficiency metric; an important concern in multi-core systems is total dissipated power. It is known that large last-level cache (LLC) is a major source of on-chip power consumption in chip multiprocessors (CMPs) because they occupy a large portion of processor die and standby power is up to 80% of their total power [25]. Recently, researchers have extensively studied the use of non-volatile memory technologies in large cache designs [26–30], in contrast to charge-based technologies (SRAM or DRAM), non-volatile memories have near-zero standby power. Among the available non-volatile technologies, resistive RAM (ReRAM) has attractive features as a replacement of SRAM in caches. Specifically, ReRAM has fast read access latency, gives about four times higher density than SRAM and is fully compatible with core fabrication process. These features make it suitable to be employed as baseline technology for LLC in deep cache hierarchies.

Compared to SRAM, write operations in ReRAM are slower and consume more power and prior work [31, 32] has mainly concentrated on alleviating the write performance and write energy issues with ReRAM. However, little attention has been paid to the problem of limited write endurance in ReRAM caches. Indeed, even though ReRAM has typically higher cell endurance (about 10^9 writes [33]) compared to competitive technologies like STT-RAM, it is still low for cache memories when write traffic of the application is high. This work studies the lifetime problem in high-capacity ReRAM caches and proposes a low-overhead and reasonable architecture to relax it.

Large caches in modern multicore processors are usually structured as *non-uniform cache architecture (NUCA)*. NUCA is a multi-bank cache where each bank is connected to one core (the number of banks is usually kept equal to the number of cores) and a switched network handles data movement between banks. NUCA caches are organized as either *static NUCA (S-NUCA)* or *dynamic NUCA (D-NUCA)*. In S-NUCA, a cache block(line) is mapped to the cache banks using a subset of bits in address and hence bank assignment is fixed. In D-NUCA, on the other hand, each cache block can be in any bank and the switched network allows data to migrate across different cache banks – that is, if a cache block is frequently used by one core, D-NUCA brings it to the local bank for future fast access. D-NUCA offers lower access latency, but it may *exacerbate the lifetime problem in ReRAM caches* because data migration between banks increases the write traffic into the cache. Moreover, if one program is highly write-intensive, it is highly probable that cache banks close to it get higher write traffic and wear out faster than others. Therefore, ReRAM D-NUCA caches may have lower lifetime than S-NUCA and sometimes lower performance in long execution of the workload.

We propose *Resistive NUCA* architecture (shortly Re-NUCA) that mitigates fast wear-out of cache banks in D-NUCA ReRAM while keeping its performance high. Re-NUCA is designed on top of *Reactive* NUCA (R-NUCA) cache [7], which is a realistic implementation of D-NUCA. R-NUCA allows data migration in NUCA but limits it to few banks close the target core (i.e., those that are only one-hop away from the local bank) and reduces the overhead of metadata required for cache

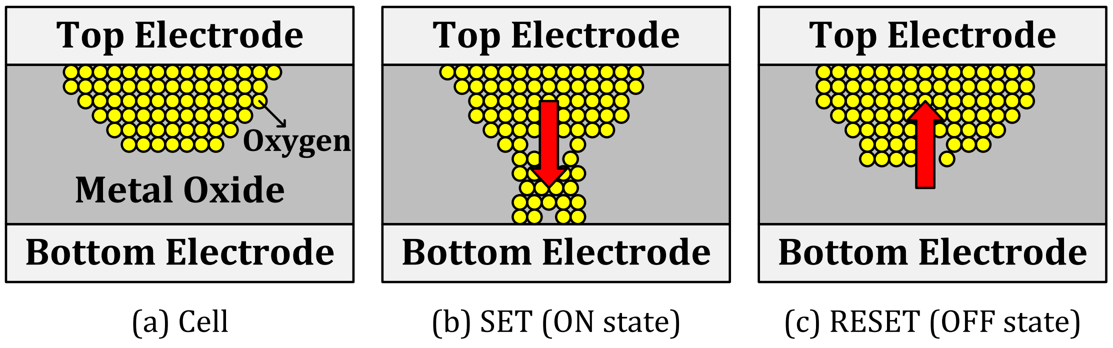


Figure 3.1: Re-RAM cell and its SET and RESET operations.

block mapping. Our proposed NUCA architecture (Re-NUCA), on the other hand, uses a “hybrid mapping function" based on criticality of the cache block[1]: *it maps the performance-critical data like R-NUCA to keep the data close to the cores, and spreads-out non-critical cache blocks to other banks (like S-NUCA).* In this way, Re-NUCA tries to reduce write intensity on cache banks by spreading them over the entire cache space, while it offers low latency by keeping the critical cache lines in the banks near to the core. Having this hybrid mapping function, Re-NUCA thus relaxes write intensity onto cache banks (almost the same as S-NUCA) while keeping performance high (close to that of R-NUCA).

Implementing Re-NUCA requires a mechanism to capture the criticality of the cache blocks. Re-NUCA determines the criticality of each cache line at the instruction level using a simple *criticality predictor* which works on the heuristics of the instruction issuing a data fetch. In addition, this architecture needs a hardware to choose proper mapping function when searching or allocating a cache line (either S-NUCA or R-NUCA mapping). Re-NUCA achieves this goal by adding few metadata bits to TLB, so (1) it reduces the overhead of this structure by avoiding to store address tag of the cache blocks, and (2) the controller of the ReRAM cache knows which function has to be used prior to access to the cache (since TLB search is performed in early cycles of memory access and the mapping information is available when accessing LLC).

This work makes the following main **contributions**:

- We propose Re-NUCA, a novel D-NUCA implementation customized for Re-RAM cache memories. It uses a hybrid of R-NUCA and S-NUCA mapping schemes, with the goal of wear-leveling the last-level caches in a performance-

[1]A cache block is assumed to be *critical*, if it contains one word (or more words) that the core needs them in short time to avoid pipeline stall.

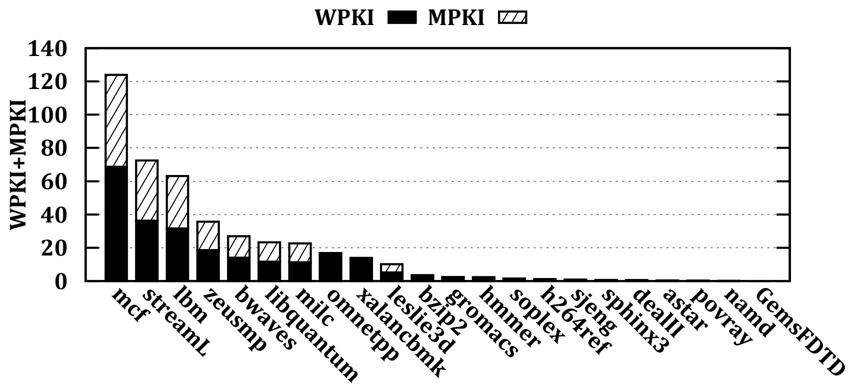


Figure 3.2: WPKI and MPKI for the studied applications.

conscious manner. Specifically, R-NUCA is used for *critical* cache blocks, and S-NUCA is used for *non-critical* cache blocks.

- To capture the criticality of a given cache line, we use a criticality predictor that determines how much a cache line is critical to the performance of the processor based on the heuristics of the instruction issuing a cache line fetch. We also suggest to keep the metadata information related to mapping function in TLB, to reduce the overhead of mapping tables and remove its access time from critical-path latency of the processor.
- We evaluate the performance and lifetime of Re-NUCA using a large set of multi-programmed workloads with different levels of memory/write intensities. Our experiments show that Re-NUCA improves the lifetime by 42%, on average, without loosing performance over R-NUCA.

3.2 Background

3.2.1 Resistive RAM

Resistive memories, in general, refer to any technology that uses a variable resistance to store information. However, ReRAM in this work refers to a subset of memories that use metal oxides as the storage medium, also called as metal-oxide ReRAM.

As shown in Figure 3.1a, a ReRAM cell consists of a metal-oxide layer sandwiched between two metal electrodes, named top electrode and bottom electrode. The cell can be either in low resistance state (i.e., SET or “1”) or high resistance state (i.e., RESET or “0”). In order to switch the state of a ReRAM cell, an external positive voltage with specific polarity, magnitude and duration has to be applied to the sandwiched layer through the electrodes. The SET and RESET operations are shown in Figure 3.1b and Figure 3.1c, respectively. When a positive biased voltage is applied to the top electrode, the metal ions (or oxygens) are forced to migrate through oxide, and eventually reach the bottom electrode. The ion-path is highly conductive, and the cell's equivalent resistance value is low (SET). The low-resistance state changes again to a high-resistance state by positively biasing the bottom electrode (RESET).

The biggest advantage of the ReRAM is its good compatibility with the CMOS process used in fabrication of logic (cores). Furthermore, the voltage required for the ion-path formation has a linear relationship with the oxide layer thickness – that is, the required voltage will decrease with a decrease in the thickness. This makes ReRAM a highly-scalable and promising alternative to SRAM. The challenging issues with ReRAM are high write latency, high write energy and low cell endurance (in terms of number of writes). A few prior studies target performance and power issues related to write operations on ReRAM when used as cache and main memory [31, 32]. This work focuses on the limited write endurance of ReRAM. Current prototypes show that a ReRAM cell can have an endurance of 10^9 [33] to 10^{11} [34–36] writes. Although this per-cell endurance is large, it can be a source of failure in cache memories when running applications are write-intensive.

3.2.2 NUCA Architecture for Large Caches

Because of small size of ReRAM cell, the on-chip ReRAM caches usually have large capacity and are subject to optimization techniques used for large caches. Large caches are usually structured as NUCA where the entire cache is partitioned into multiple banks. Each bank is connected to one core and an on-chip network for data and address transfer between banks. NUCA exhibits varying access latencies depending on the distance between the data and the core requesting it. In static NUCA (S-NUCA), mapping a cache block to banks is fixed and is determined using the lower bits of the block's address. This makes redirection (finding the target bank on requesting a cache block) in S-NUCA simple and obviates the need for any lookup table. In D-NUCA, any given line can be mapped into several banks, and a cache block can migrate between banks according to the access frequency – that is, frequently used cache lines migrate to banks closer to the core. D-NUCA needs a table to keep the redirection data for each cache block that is kept along with coherency information in the directory. On a cache access, the directory is checked for coherency issues, and if it is a hit, the associated redirection information is also read to determine the bank index currently holding the cache block.

Accessing D-NUCA for this metadata information increases the energy consumption of the directory and increases the traffic of the switched network. This clearly complicates the implementation of D-NUCA. Considering the overheads of migration to keep the data close to the core requesting the data like in D-NUCA, Hardavellas, et al. proposed *Reactive* NUCA (R-NUCA), [7] which tries to combine the benefits of both S-NUCA and D-NUCA. In R-NUCA, cache blocks for each core are allowed to be stored in a fixed-size cluster that includes banks that are (at most) one hop away from the core. Figure 3.4(a) pictorially shows bank-level clustering in R-NUCA for a cache with 16 banks and 16 cores. In this example, the cache blocks requested by core are allocated in the shaded region. As can be observed, shaded region cache banks are at most one hop away from the requested core. Thus cache lines accessed by each core are always close to it (at most one hop away from the target core), which resembles D-NUCA in performance. Similar to S-NUCA, the address redirection in each cluster is done by simply decoding few low-order bits for the bank index. The mapping function used by R-NUCA is:

$$DestinationBank = (Addr + \overline{RID} + 1) \& (n - 1),$$

where RID is the rotational ID in [7] and n is the cluster size which in this case is 4.

3.3 Application characteristics and motivation for wear-leveling

Figure 3.2 plots the Writebacks Per Kilo Instruction (WPKI) and Misses Per Kilo Instructions (MPKI) for different applications used in our evaluation. As writes to the L3 caches come from both write backs from L2 and a cache line fetch upon a L3 miss, Figure 3.2 shows the LLC write intensity of various applications when an application runs individually with a 256KB L2 and 2MB L3 caches. Hence there is wide diversity among applications's lastlevel cache write and miss characteristics.

To demonstrate the importance of NUCA architecture and its mapping scheme on the write count distribution over cache banks, we performed a series of lifetime and performance evaluation in a system with the configuration given in Table 7.1. The L3 cache (last-level cache) is made up of ReRAM, has 16 banks each with 2MB size, and has the NUCA structure with 4×4 on-chip network between cache banks. The system has 16 cores and runs a workload of 16 single-threaded application from the SPEC CPU 2006 suite [37]. Our multi-program workloads include applications with diverse write intensities and the number and pattern of writes can vary greatly from core to core, depending on the application.

Figure 3.3 shows the lifetime variation between banks of the L3 cache for different NUCA architectures, over all evaluated workloads. We evaluated *S-NUCA*, *R-NUCA*, *private cache* (each core has a 1MB private L3 cache) and a cache architecture with perfect wear-leveling scheme (named *Naive* and discussed later in this section). The numbers presented in y-axis are the harmonic mean of lifetimes across all the workloads that is calculated as follows: we run 10 workloads of varying memory intensities and calculate the lifetimes experienced by the cache bank over all these workloads. The harmonic mean lifetime of a cache bank is the harmonic mean of these lifetimes. As discussed earlier, S-NUCA evenly stripes the memory space across all cache banks in the system, so that every core will access all the banks. We expect writes being more uniformly distributed over cache banks in S-NUCA, when compared to two other designs (R-NUCA and Private). The results plotted in Figure 3.3 confirm this finding, as all cache banks have very similar lifetime in the S-NUCA architecture, regardless of the memory-intensity of the workload bound to each core. The other extreme design is private cache for each core that offers maximum variation in lifetime of the cache banks – that is, the most heavily written cache bank has a lifetime of less than 2 years in our experiments. We also observe that R-NUCA has relatively large variation between lifetime of the cache banks. The reason is that, in R-NUCA, since data blocks of a core are concentrated in a cluster of four banks (compared to a single LLC in the case of private caches), the clusters of banks used by the memory-intensive applications will still wear out more quickly than the clusters used by the low memory- and write-intensive applications.

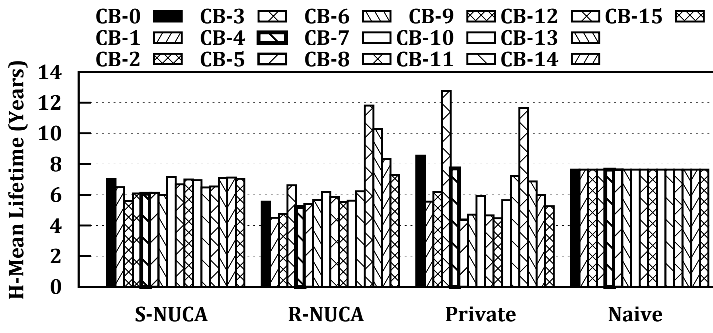


Figure 3.3: Harmonic mean lifetime in years. CB refers to a cache bank.

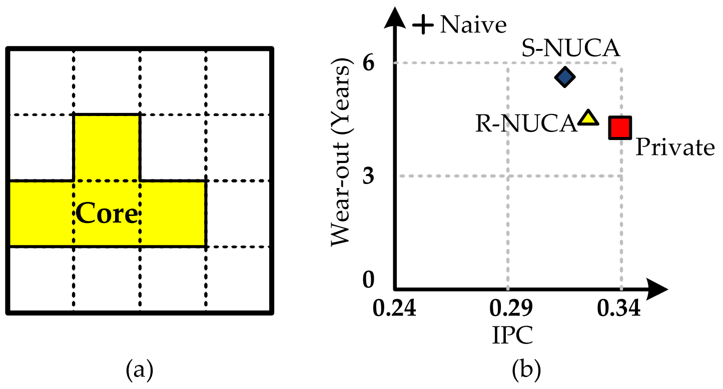


Figure 3.4: (a) The shaded region is R-NUCA region for a given core, (b) Comparison of performance and wearout characteristics of different cache architectures.

3.3.1 Perfect (Naive) wear-leveling approach

A performance-agnostic perfect wear-leveling approach would wear-level the cache banks perfectly by ensuring that each cache bank would receive the same number of writes. Such a perfect wear-leveling scheme allows us to compare how well a NUCA scheme performs with respect to cache bank wear-leveling. This scheme needs oracle knowledge about the number of writebacks and misses incurred for every cache bank. Apart from the oracle knowledge about the individual cache bank, this scheme would also require a directory to know which cache bank contains a particular cache line for a cache line look up after a miss in the L2 private cache. The directory overhead for a high capacity last-level cache is significant and hence this scheme is not a feasible option in a commerical processor and, in this work, we use it just for comparison. We interchangeably use “Naive” to refer to this perfect wear-leveling scheme as it naively accounts only for the lifetime of cache and ignores the performance.

In our implementation, we keep track of the total number of LLC misses and writebacks (i.e., total writes to the cache) for each bank. When a new cache line needs to be written into the cache, the cache controller chooses the bank with the smallest number of writes so far. This approach leads to near-ideal wear-leveling as shown in Figure 3.3, with 0% variation in lifetimes between banks. However, as this scheme does not consider performance while wear-leveling, it degrades the application performance by 21%, on average, compared to S-NUCA.

3.3.2 Performance versus lifetime of various NUCA schemes

Figure 3.4 shows the trade-off between performance and lifetime for various cache architectures: *S-NUCA*, *R-NUCA*, *Private* and *Naive*. The numbers presented in Figure 3.4 are the harmonic mean values of 10 workloads, which are explained in detail in Section 7.6.2. In Figure 3.4, y-axis represents the lifetime in years which signifies the number of years beyond which we loose the whole cache capacity, and therefore, a higher number on that axis is better. X-axis represents the Instructions Committed Per Cycle (IPC). A higher IPC value means a higher performance; hence, a larger number on X-axis is better. As can be observed from this figure, the Naive mapping policy, which balances the number of writes and misses achieves a maximum lifetime of more than 6 years, does not fare well in terms of performance. Note also such a perfect mapping policy is not practical as we explained above. While the Naive mapping scheme performs best for lifetimes, private cache banks perform best for performance. However, private caches perform worst in terms of lifetimes as the writes are distributed unevenly across the cache banks. The next best scheme, as can be observed in Figure 3.4(b), for better wear-leveling is S-NUCA. Since S-NUCA uses address bits to interleave cache lines across various cache banks, it is not a better scheme for performance as it incurs on-chip traffic for accessing cache blocks that are interleaved across the cache banks. Reactive(R)-NUCA proposed in [7] perrforms close to private caches in terms of performance and fares slightly better than private caches in terms of lifetimes. We remark that Even though R-NUCA is good in performance, with time, cache banks wear out and we loose cache capacity without wear-leveling in place thereby hurting the performance.

From above, there is a clear necessity for a new NUCA architecture which fares well in terms of *both* performance and wear-out. In the next section, we propose a NUCA architecture and related policies to achieve this goal.

3.4 Re-NUCA Architecture

Ideally, a wear-level and performance-aware NUCA cache should place all the important cache lines in the nearby cache banks, and spread out all the non-important cache lines across various cache banks. Important data cache blocks are often referred to in the architectural community as *critical cache blocks* and loads that fetch such critical cache blocks are referred to as critical loads. Our proposed architecture, Resistive NUCA or Re-NUCA, allocates the critical cache blocks closer to the core running the application in a region called the Re-NUCA region, while spreading out the cache blocks that are not critical to the performance using S-NUCA mapping. By spreading out non-critical cache blocks using S-NUCA, write-backs to such non-critical cache blocks can also be distributed across cache banks.

When a cache line is brought to the cache for the first time, we assume a cache-line is not critical, hence a cache line is mapped using S-NUCA. This presumption helps us in prioritizing lifetime over performance for a cache line. Later, based on the output of the criticality predictor logic, the decides on the mapping policy used for cache line allocation. In the following we describe data criticality and the logic we used for criticality prediction.

3.4.1 Critical data and criticality predictor

To explain our notion of criticality better, it is important to consider the micro-architecture of current processors. Most of the commercial processors available in the market currently perform out-of-order execution to achieve maximum performance. Even though instructions are executed out-of-order in the processor, they are committed in-order. All out-of-order processors typically employ a special hardware structure called ReOrder Buffer (ROB), shown in Figure 3.6(a). The ROB contains all the instructions that are being executed, and the instructions at the head of the ROB are committed upon execution. If the instruction at the head of the ROB is not executed, head of the ROB is stalled until the instruction is executed.

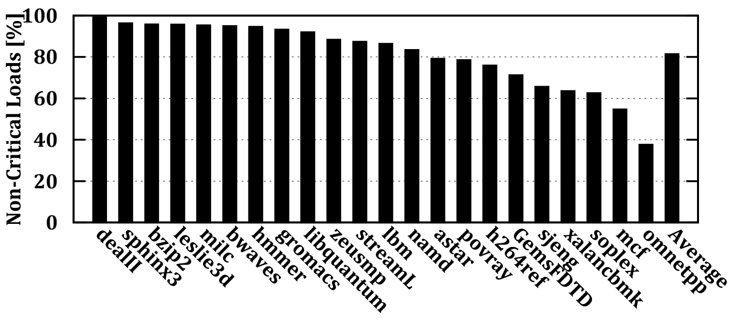


Figure 3.5: ROB stall percentage.

A load issued by a processor is considered critical if it *blocks* the head of the ROB. As explained above, since out-of-order processors follow in-order commit, all the other instructions, which are executed and are ready to be committed, are stalled by the blocking load. As a result, a load which stalls the head of the ROB prevents other ready instructions from being committed thereby decreasing the performance of the application. Such loads that block the head of the ROB are defined as critical loads.

Every instruction executed on a processor contains a Program Counter (PC) which is unique for a specific instruction. Since many data-intensive applications spend considerable amount of time executing loops, each instruction in the program is executed multiple times in different iterations on different data. Consequently, the PC can serve as a valuable attribute to predict various properties of a program phase. Prior works [38] have used this PC to predict the criticality of a load. In this work, we employ a slightly modified version of the criticality predictor presented in [38], as described later in this section.

Figure 3.5 shows the percentage of loads that do not stall the head of ROB for various SPEC CPU2006 benchmarks. For these experiments, each benchmark is executed on a 2.4 GHz out-of-order processor containing a private L1(32KB), L2(256KB) and L3(2MB) caches with a DDR-3 memory channel. On an average, over 80% of all loads issued by the processor do not stall the ROB, meaning that non-critical loads contribute to 80% of loads. These results represent a promising direction to identify cache blocks that do not result in performance degradation. That is, a criticality predictor which can accurately predict the non-critical loads is important to identify which cache blocks can be spread over the other banks without incurring any performance loss.

3.4.2 Criticality Predictor

Our criticality predictor adapts hardware structures similar to the Commit Block Predictor presented in [38], which we refer to as the *Criticality Predictor Table* (CPT). More specifically, it contains the following counters:

(1) a PC associated with a load instruction,

(2) a counter similar to robBlockCount in [38] that denotes the *number of times* the ROB has been blocked by the corresponding PC in the past.

(3) a counter numLoadsCount, that indicates the number of loads that were issued by this PC up to this point.

Figure 3.6(b) shows the operation of the criticality predictor. When a load is issued by the processor, the CPT is indexed with the PC as shown in step 1, and if an entry for the corresponding PC exists, numLoadsCount is incremented, which is referred to in Figure 3.6 as step 2. If this load results in ROB head block, the robBlockCount counter is incremented which is referred to as step 3. If the CPT does not contain an entry with the corresponding PC, a new entry with the corresponding PC will be inserted into the CPT when the load is committed. The new entry will have numLoadsCount counter set to 1 and a robBlockCount of 1 or 0, depending on whether the load results in a ROB head stall or not.

The difference in our criticality predictor compared to the one proposed in [38] is that we do not need additional information for a PC viz, *LastStallTime*, *MaxStallTime* and *TotalStallTime* as proposed in [38] since we do not have to rank the loads in terms of criticality. Hence, our scheme does not incur any additional overhead as it just needs an extra bit to be sent to the mapping logic to identify if a load being issued is critical or not and averts all the complexity involved in tracking the stall cycles and the corresponding storage.

When a load is issued by the processor, if the CPT lookup results in a hit, we read the robBlockCount and numLoadsCount from the CPT. If the robBlockCount is greater than or equal to a threshold, $x\%$ of the numLoadsCount, we mark the load as a critical load. This threshold x, referred to as the “criticality threshold" in this work, determines the accuracy of the criticality predictor. For example, if the value of x is 100%, then we predict a load as critical if 100% of the load instructions issued by that PC in the past have resulted in a ROB stall. In general, our experiments reveal that a smaller criticality threshold leads to better criticality

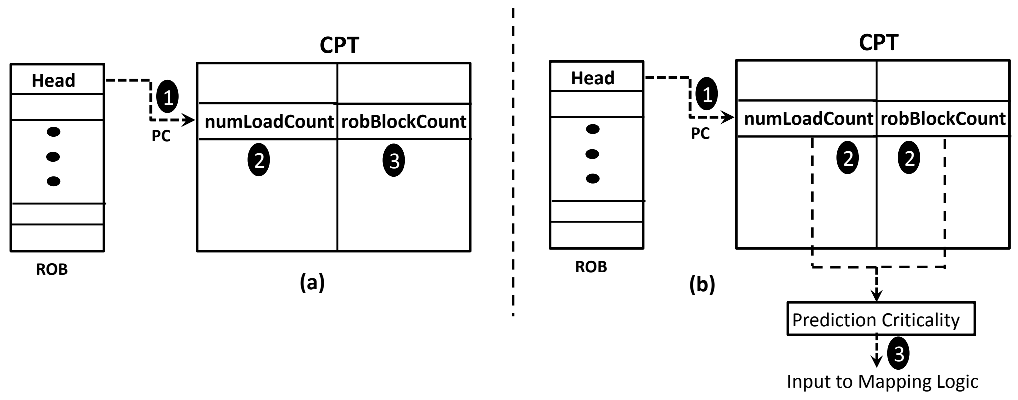


Figure 3.6: (a) CPT update, and (b) CPT lookup.

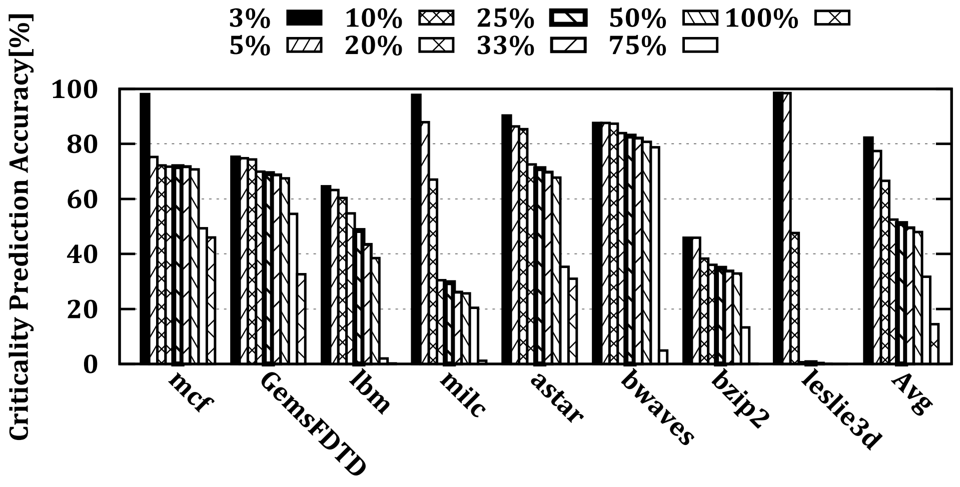


Figure 3.7: Criticality prediction accuracy.

predictions. A criticality threshold of 100% is a stringent condition. The accuracy of our criticality predictor for different thresholds is plotted in Figure 3.7. As can be observed, a high criticality threshold of 100% results in a lower accuracy of 14.5%, while a criticality threshold of 3% results in an accuracy as high as 83%, on average with the maximum being around 99% for a workload. Based on this result, we use 3% as our criticality threshold.

Figure 3.8 shows the percentage of non-critical cache blocks with various criticality threshold values. It shows that, around 50.3% of the cache blocks are fetched from memory are non-critical and do not result in any ROB stall. Figure 3.9 plots the number of writes to the non-critical cache blocks identified through our criticality predictor. With a criticality threshold of 3%, we observe that around

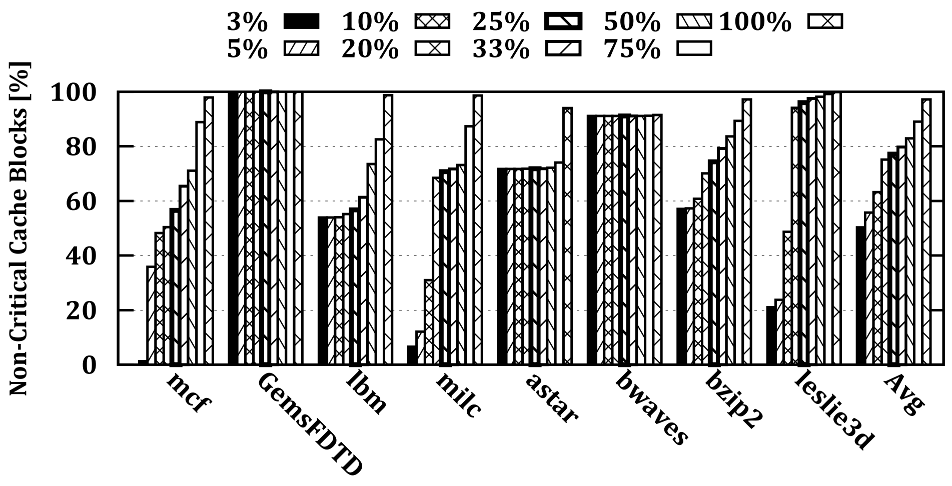


Figure 3.8: Percentage of noncritical cache blocks.

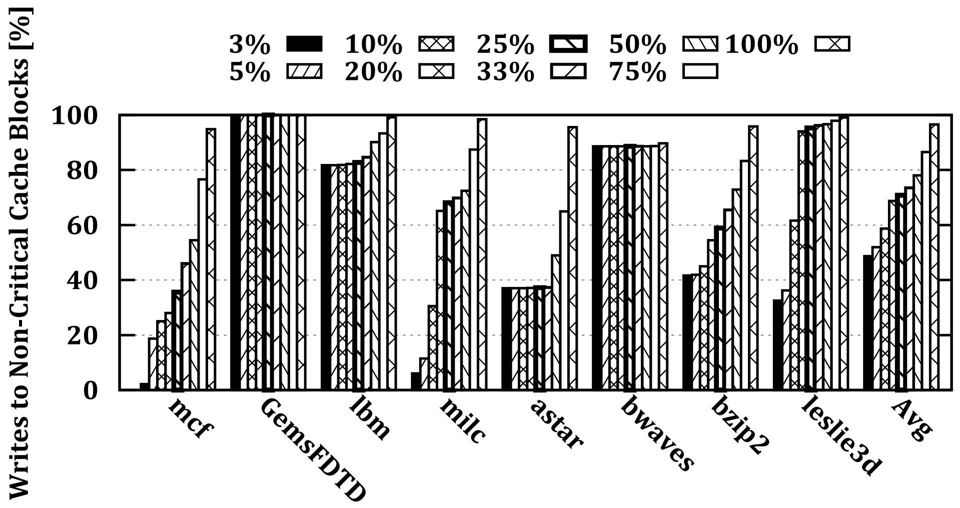


Figure 3.9: Writes to noncritical blocks.

50% of the writes go to non-critical cache blocks and, as a result, these writes can be distributed across various cache banks to reduce the wear-out without causing the performance to degrade. The non-critical loads shown in Figure 3.5 are different compared to that in 3.8 as the non-critical loads in Figure 3.5 accounts multiple loads to the same cache blocks as critical or non-critical and hence will account to the on-chip cache hits as well. However, the non-critical loads in Figure 3.8 just accounts for the loads that result in a on-chip cache misses and a memory access, which present an opportunity for cache wear-leveling. Hence, even though Figure 3.5 indicates 80% non-critical loads on an average, since our mechanism

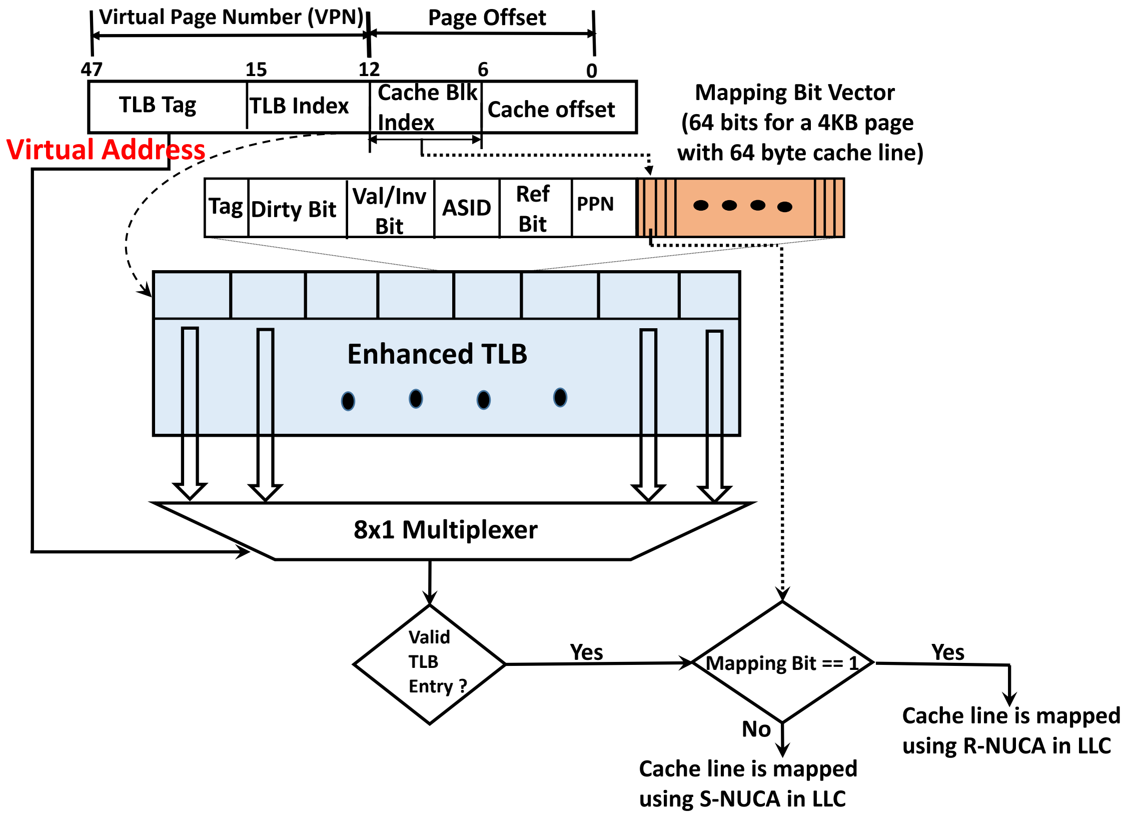


Figure 3.10: Enhanced TLB with 64 entries, each 8-way showing Mapping Bit Vector for each entry.

does not consider migration of cache blocks, we can only act on 50.3% of the cache block loads, as shown in Figure 3.8.

3.4.3 Enhanced TLB

Since a Program Counter (PC) can change from being non-critical to critical and vice versa over the course of execution, we need to store the mapping information for the cache lines that have already been allocated using one of either S-NUCA or R-NUCA mappings. Upon a miss in the private L1 and L2 caches, the mapping information is used to lookup the corresponding cache bank. We propose an enhanced TLB architecture, which augments a conventional TLB with this mapping information. Every load and store instruction go through the TLB to obtain the physical address of the requested cache line. TLB ideally contains the virtual address-to-physical address translation for a page (typically 4KB), and hence contains information at coarse granularity. Assuming cache line to be 64 bytes, each 4KB page contains 64 cachelines. Since a cache line to be accessed in a page is already part of the virtual address, we can use the same virtual address bits to lookup the TLB entry. In our enhanced TLB architecture, TLB is augmented with a Mapping Bit Vector (MBV) of length 64 bits, each bit corresponding to a cache line in the page. The proposed architecture for Enhanced TLB is shown in Figure 3.10. The cache line index bits from the virtual address are used as an index into the MBV to update or read the mapping information of a particular cache line.

Upon a last-level cache miss for a cache line, based on the predicted criticality, the corresponding cache line is allocated in cache bank. Once the data is returned to the processor, MBV in the TLB for the corresponding cache line is updated. If the cache line is mapped in the last-level cache bank using S-NUCA (non-critical), then the MBV bit is set to 0; else it is set to 1, indicating that it uses R-NUCA (critical). In our proposal, since a cache line does not change the criticality status in its on-chip lifetime, we do not need to update the MBV bits for a cache line unless the cache line is to be evicted. When a cache line is being evicted, the corresponding MBV bit needs to be reset back to 0.

Our enhanced TLB contains 64 entries in both L1I and L1D per core. Each of them is 8-way set-associative. As each entry in the enhanced TLB contains an extra 64 bits in the MBV, our proposed enhanced TLB architecture adds an extra over-head of 1KB per core, that is 512 bytes for L1I and 512 bytes for L1D TLBs. For a 16 core processor, our enhanced TLB architecture requires an extra 16KB storage, which is less than a single L1I/L1D cache size. Hence, the overhead of our enhanced TLB architecture is negligible in terms both area and power.

We want to emphasize that Re-NUCA tries to achieve the best of both S-NUCA and R-NUCA by combining the performance benefits of R-NUCA by allocating only the critical cache blocks closer to the processor, and wear-leveling of S-NUCA by spreading out the non-critical blocks.

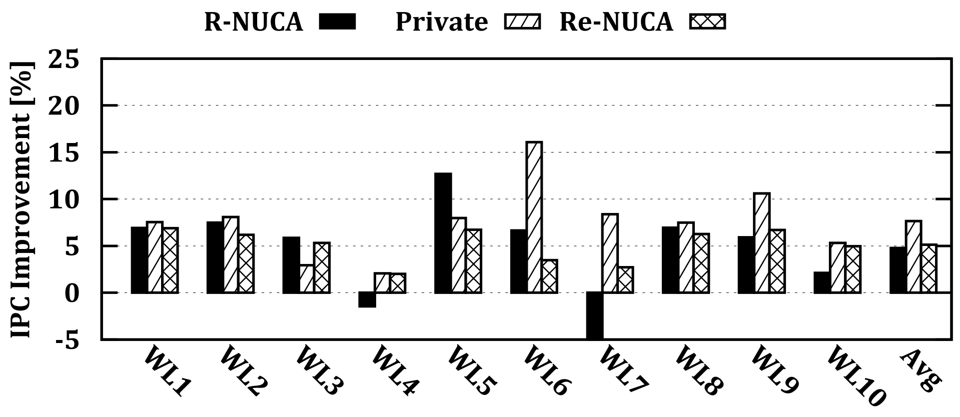


Figure 3.11: IPC Improvements. All the improvements are normalized to S-NUCA.

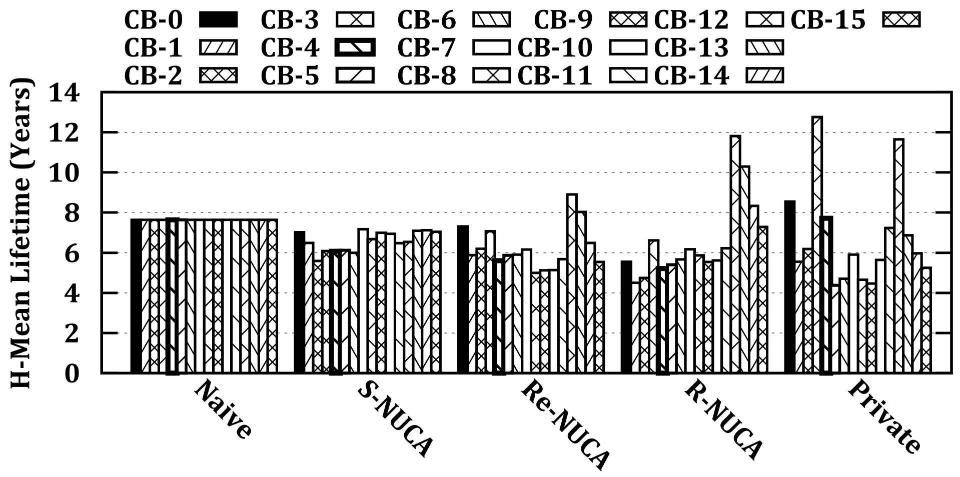


Figure 3.12: Re-NUCA wearout.

3.5 Experimental Evaluation

3.5.1 Evaluation Environment

In this work, we used GEM5 [39] to evaluate our Re-NUCA. We used the system-call emulation (SE) mode of GEM5 instead of the time-consuming full-system (FS) mode. SE mode of simulation in GEM5 is faster compared to the full-system mode. Table 7.1 gives our target multicore system. We fast-forward around 2 billion instructions for each benchmark to get to the region of interest, warmup the caches by running 100 million instructions for each benchmark, and then simulate the next 100 million instructions on each core to collect the statistics. We consider

Cores	16 cores @ 2.4GHz, ALPHA ISA, out-of-order
ROB entries	128
NoC	4x4 Mesh
L1I/L1D Cache	32KB, 4-way associative, 2-cycle latency, 64 Bytes cache line
L2 Cache	256KB (private), 8-way associative, 5-cycle latency, 64 Bytes cache line
L3 Cache	2MB per core, 32MB total, 16-way associative, 100 cycle latency, 64 Bytes cache line
Cache Coherence	MESI Protocol
Memory	JEDEC-DDR3, 16GB DRAM, 4 channels, 2 ranks per channel, 8 banks per rank, FR-FCFS memory scheduler

Table 3.1: Simulated architecture configuration.

Application	**WPKI**	**MPKI**	**Hitrate**	**IPC**	**Application**	**WPKI**	**MPKI**	**Hitrate**	**IPC**
mcf	68.67	55.29	0.20	0.07	omnetpp	16.22	0.61	0.96	0.78
streamL	36.25	36.25	0.00	0.37	xalancbmk	13.17	0.76	0.94	0.89
lbm	31.66	31.46	0.01	0.53	leslie3d	5.24	4.86	0.07	1.33
zeusmp	18.57	17.13	0.08	0.54	bzip2	2.89	0.69	0.76	1.63
bwaves	14.01	12.91	0.08	0.59	gromacs	1.85	0.61	0.67	1.61
libquantum	11.67	11.64	0.00	0.34	hmmer	2.20	0.13	0.94	2.61
milc	11.31	11.28	0.00	0.71	soplex	1.27	0.25	0.80	0.94
GemsFDTD	0.00	0.01	0.00	1.81	h264ref	1.09	0.08	0.93	2.00
namd	0.04	0.05	0.21	2.34	povray	0.18	0.04	0.79	1.57
astar	0.24	0.12	0.54	2.08	dealII	0.33	0.12	0.65	2.27
sphinx3	0.30	0.30	0.06	1.96	sjeng	0.52	0.32	0.41	1.16

Table 3.2: Applications used in the experiments. IPC values shown are for a single core.

ReRAM cache line to wear out beyond 10^{11} writes.

We used the SPEC CPU 2006 benchmarks with their reference inputs. Table 7.2 presents various characteristics of these applications like IPC, last-level cache hit rates, last-level Writes Per Kilo Instruction (WPKI), and last-level cache Misses Per Kilo Instruction (MPKI). As can be observed from this table, these applications exhibit quite a variation in performance; some are memory intensive while others are compute intensive. Based on the sum of the WPKI and MPKI values shown in Table 7.2, we characterize our applications as high, medium and low write intensive. Applications with sum of a WPKI and MPKI greater than 10 are categorized as high write-intensive ; applications with a sum between 1 and 10 are categorized as medium write intensive while all the applications having a sum less than 1 are categorized as low write intensive ones.

We further formed 16-core workloads by randomly choosing applications from the high write-intensive ones along with the medium- and low-intensive ones. Since the cache endurance problems mostly occur when high write-intensive applications are run and the imbalance in wearout occurs when the high write-intensive applications

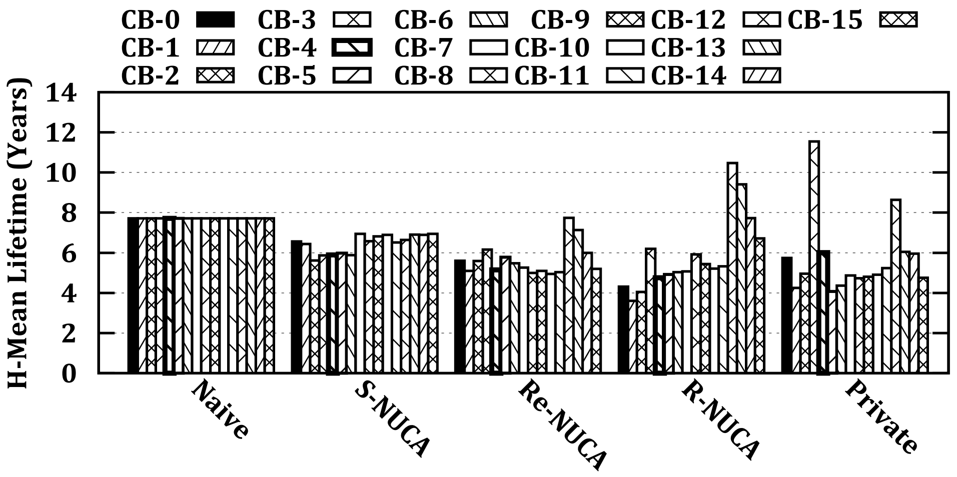


Figure 3.13: Wear-leveling with an L2 size of 128KB.

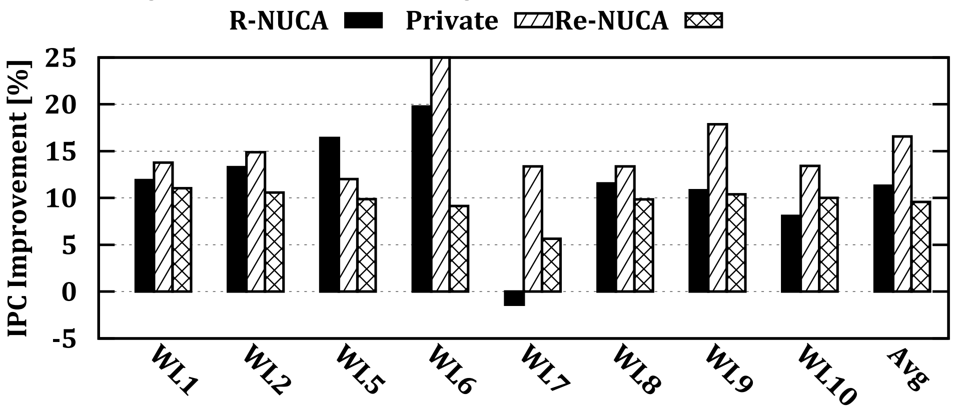


Figure 3.14: IPC improvements with L2 size of 128KB.

are run with the medium- and low-intensive applications, we choose workloads such that we always run high memory-intensive applications with low/medium write-intensive applications.

3.5.2 Results

We present liftime and performance results for Naive, S-NUCA, R-NUCA, Private and compare them with our proposed Re-NUCA scheme. We report the following results across all the NUCA schemes: harmonic lifetime in years, IPC, and raw minimum lifetime in years. Harmonic lifetime in years represent the harmonic mean of lifetime in years across all the workloads per cache bank. Harmonic mean of lifetimes is a better compared to an average as average lifetime is significantly

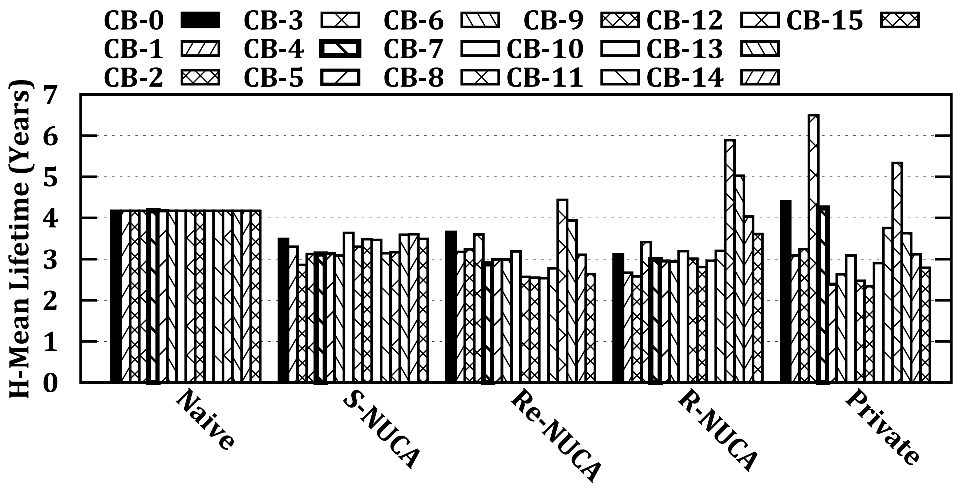


Figure 3.15: Wear-leveling with an L3 size of 1MB.

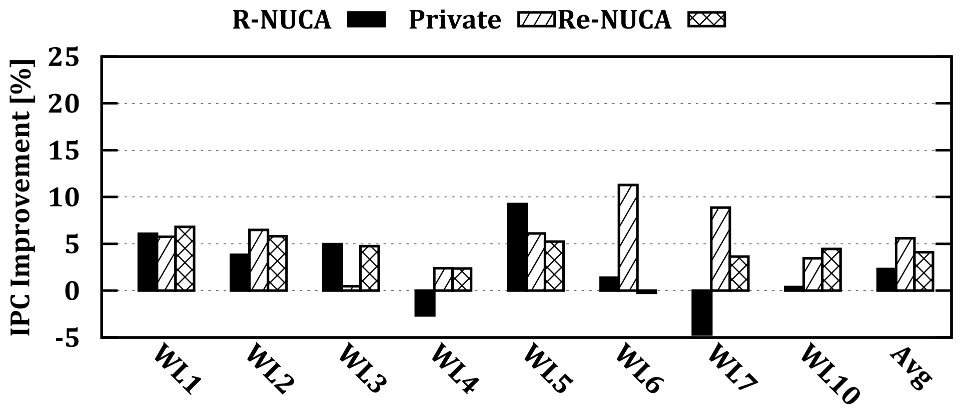


Figure 3.16: IPC improvements with L3 size of 1MB.

effected by the extremes. On the contrary the raw minimum lifetime gives the minimum lifetime of any cache bank in all the workloads. This metric helps us to observe how much one can improve the lifetime of a cache bank, which is under write pressure, under different NUCA configurations. Note that higher values of harmonic and raw minimum lifetimes are better. We use a metric called Instructions committed per cycle (IPC) to evaluate the performance of processor for each NUCA scheme. IPC is an accurate metric for multi-programmed workloads and gives the throughput of an out-of-order processor. Higher values of IPC are better.

Figure 3.12 shows the harmonic mean lifetimes for Re-NUCA mechanism compared to the other mechanisms. X-axis in Figure 3.12 represents various NUCA schemes, and Y-axis represents the harmonic lifetime in years. Hence, higher the number on Y-axis the better.

The Naive mechanism which does not consider performance and tries to wear-level the cache banks result in the best harmonic lifetime of around 7.5 years. Also, the variation of lifetimes across all the cache banks is 0, and thus Naive mechanism gives the best wear-leveling possible. One can observe from Table 3.3 (Actual Results row) that Naive scheme also has the best raw minimum lifetime of 5 years. The Naive scheme degrades the performance on average by 21% compared to S-NUCA. Hence, all the IPC improvements are normalized with respect to S-NUCA as the Naive scheme performs worse even in the sensitivity analysis. This can be attributed to the fact that the Naive mechanism does not consider criticality of cache blocks while choosing the destination of the cache blocks. At the other end of the spectrum is the private cache configuration which has the worst raw minimum lifetime of 2.3 years, as seen in Table 3.3 (Actual Results row) for Private, and huge variations in the harmonic lifetimes across cache banks as can be observed in Figure 3.12.

The private cache configuration, as explained before, localizes writes and misses to the corresponding cache bank without spreading the writes/misses unlike other NUCA schemes. Consequently, a high memory and write intensive application like mcf wears-out its own last-level cache bank faster than other (non-memory and non-write intensive) applications. Also, since the private cache configuration does not warrant on-chip traffic for last-level cache hits, the performance for private cache configuration is the best compared to the other NUCA schemes tested as can be observed in Figure 3.11. However, the private cache configurations suffer from the capacity utilization problem as the last-level caches are not shared. This is the reason why we see that IPC is lower in some workloads in Figure 3.11 compared to R-NUCA. However, the private cache configuration does not suffer from cache-interference, a problem with the shared last-level caches. As a result, private caches incur high IPC in most of the workloads compared to other NUCA schemes and on an average achieves around 8% improvement in IPC compared to S-NUCA and around 4% improvement compared to R-NUCA. The improvements are as high as 16% compared to S-NUCA and 14% compared to R-NUCA for certain workloads, as can be observed in Figure 3.11.

As the Naive and private cache configurations fall at the two ends of the spectrum, the other NUCA schemes S-NUCA and R-NUCA have mediocre IPC and harmonic/raw lifetime compared to the Naive and Private cache configurations.

Application	**Naive**	**S-NUCA**	**Re-NUCA**	**R-NUCA**	**Private**
Actual Results	4.95	3.37	3.24	2.38	2.32
L2-128KB	7.14	3.9	3.09	2.31	2.31
L3-1MB	3.64	1.67	1.67	1.38	1.38
ROB-168	7.06	3.26	3.26	2.33	2.32

Table 3.3: Raw Minimum lifetimes.

While S-NUCA contains better harmonic/raw minimum lifetime compared to R-NUCA, R-NUCA gives better performance compared to S-NUCA. On an average, R-NUCA beats S-NUCA by 4.7% in IPC, and S-NUCA has a better raw minimum lifetime of 3.36 years while R-NUCA contains a raw minimum lifetime of 2.38 years. The maximum performance difference between R-NUCA and S-NUCA is 12.8% for a certain workload (Figure 3.11).

Our Re-NUCA scheme, as can be observed in Figures 3.11, 3.12 and in Actual Results row for Re-NUCA in Table 3.3, retains the best of both worlds from S-NUCA and R-NUCA in terms of the raw minimum lifetime and performance. By placing the critical blocks closer to the target core in the Re-NUCA region, our Re-NUCA configuration achieves a performance improvement of 5.2% on average, and up to 6.9% for a workload compared to S-NUCA and equalling the performance of R-NUCA on average. On the other hand, by spreading the non-critical cache blocks using S-NUCA approach, *our Re-NUCA scheme wear-levels the cache in a performance-conscious fashion.* The raw minimum lifetime of Re-NUCA scheme is 3.24 years bettering the R-NUCA raw minimum lifetime by 42%, as can be seen in Table 3.3 while retaining it's performance. Once can see from Figure 3.12 that Re-NUCA wear-levels the cache banks in R-NUCA by increasing the harmonic lifetime of cache banks with lower lifetime such as cache banks cb-0, 1, 2, 4, 5, 6, while reducing the harmonic lifetime of other cache banks which are remarkably high such as cache banks, cb-12, 13,14 and 15. Recall that Re-NUCA employs a hybrid mechanism of R-NUCA and S-NUCA in a performance-conscious manner. If a cache block is predicted as critical, it is allocated in the Re-NUCA region close to the target core, whereas if the cache block is predicted as non-critical, cache blocks are spread-out across last level cache banks using S-NUCA.

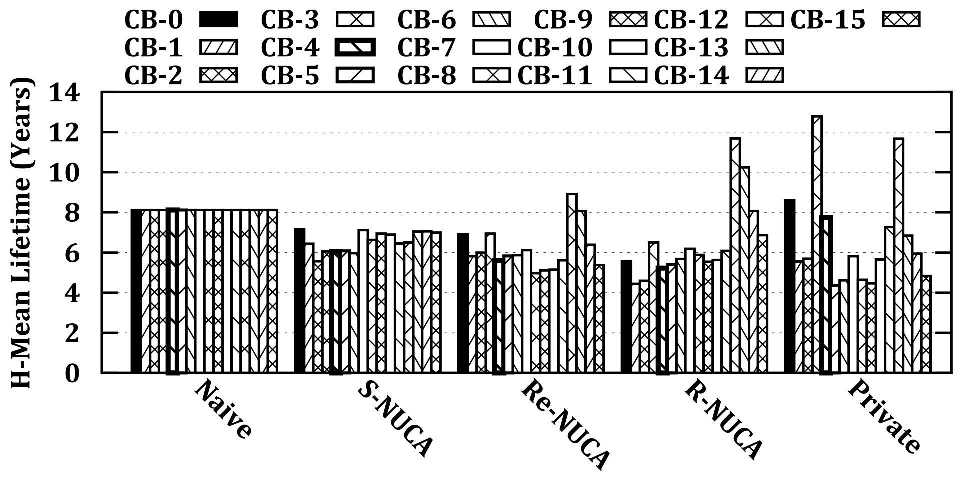


Figure 3.17: Wear-leveling with an ROB of 168-entry size.

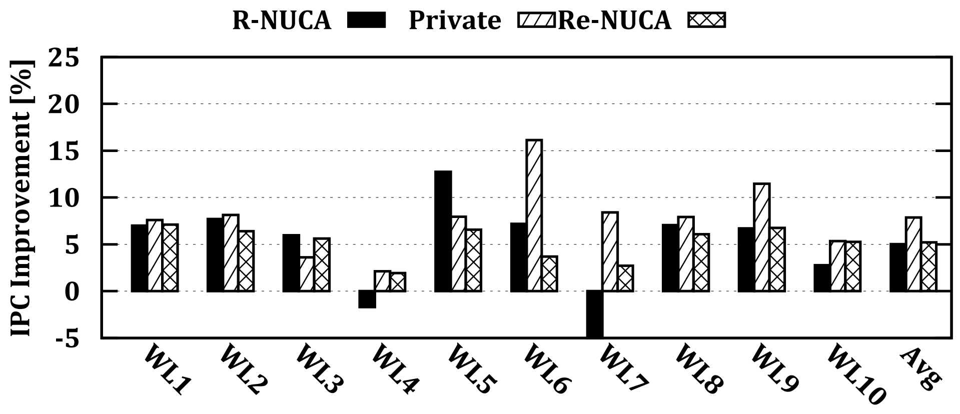


Figure 3.18: IPC improvements: with an ROB size of 168.

3.5.3 Sensitivity Analysis

In this subsection, we discuss how our Re-NUCA works with varying sizes of caches in the cache hierarchy and with the changes in the micro-architecture of the processor especially the number of ROB entries. One of the cache hierarchy parameters which effect the number of writes to the last level cache banks is the sizes of L2 and L3 cache banks itself. Since reducing the size of L2 increases the number of L2 misses and hence the write-backs, we evaluated the impact of Re-NUCA by decreasing the size of L2 to 128KB while our default system had 256KB. Figure 3.13 and the L2-128KB row in Table 3.3 show the harmonic and raw minimum lifetime of Re-NUCA compared to the other schemes. As can be observed, Re-NUCA reduces the variation in harmonic lifetimes across the cache banks compared to R-NUCA, while managing the performance degaradation of only 1.5%, on an average, compared to R-NUCA as can be observed in Table 3.3 in L2-128KB row. The raw minimum lifetime of Re-NUCA with 128KB L2 is 3.10 years compared to R-NUCA with 2.3 years thereby improving the raw minimum lifetime by 34.8%.

As our next sensitivity experiment, we reduce the size of L3 Re-RAM cache bank to 1MB while our baseline consists of 2MB per cache bank. With the decreased L3 cache bank size, the memory-intensive application incurs more L3 cache misses, which results in fetching more cache blocks and thus increasing the writes to the L3 cache bank. The harmonic mean and lifetime improvements are plotted in Figures 3.15 and in L3-1MB row in Table 3.3, respectively. As can be observed, Re-NUCA wear-levels the lifetimes similar to L2 with 128KB case. Re-NUCA improves the raw minimum lifetime compared to R-NUCA from 1.38 to 1.67 years improving it by 21%. On average, Re-NUCA improves performance compared to R-NUCA by around 1.8%. It also increases performance by 4.11% compared to S-NUCA on an average. However, the maximum improvements acheived by Re-NUCA compared to R-NUCA and S-NUCA observed are 8.2% and 6.81% respectively.

The other micro-architectural characteristic that influence the impact of Re-NUCA is the number of entries in the ROB itself. With the increased number of entries in the ROB, some of the loads might endup not stalling the ROB, thereby effecting the criticality predictor. Next, we conducted experiments by increasing the number of entries in ROB to 168, while in our baseline configuration ROB contained 128 entries. With increased ROB size, as can be observed in Figure 3.19, IPC improves by 5.2% compared to S-NUCA, while it is slightly better than R-NUCA by 0.5%. However, the raw minimum lifetime improves from 2.33 years to 3.26 years compared to R-NUCA, improving its lifetime by around 39.9%, as opposed to 42% with an ROB containing 128 entries.

3.6 Related Work

EqualChance by Mittal and Vetter [40] moves write-intensive cache blocks to a different set in a cache. It keeps track of the writes per set and redirects write to another clean or invalid location if the number of writes goes over a threshold. i2wap (Wang et al.) [41] is a cache management strategy that balances writes between

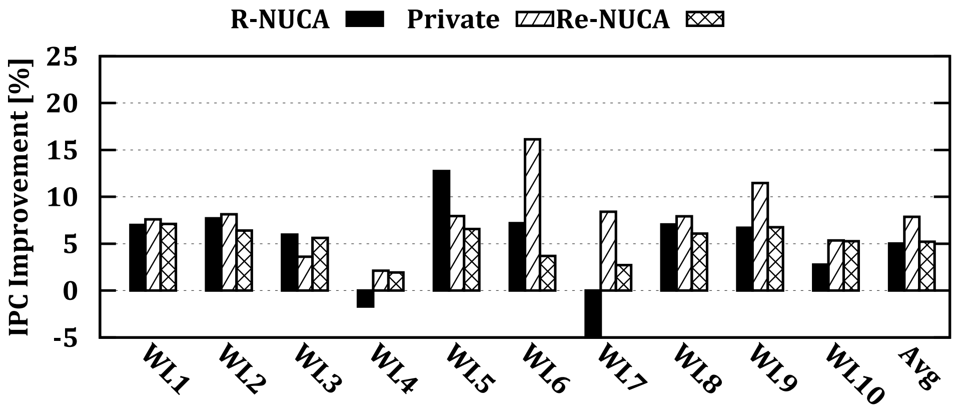


Figure 3.19: IPC improvements: with an ROB size of 168.

sets and within a set. They combine a main memory wear-leveling technique for addressing variations between sets using a technique to reduce variation within a set.

In our work, we target inter-cache bank wear-leveling while the works mentioned above try to wear-level a cache bank at a finer granularity at inter-set and intra-set level. Though our approach is orthogonal to their approaches, their approaches can be complementarily implemented on top of our proposed approach to reap higher benefits in terms of wear-leveling in a performance-conscious fashion.

Chapter 4

Congestion-Aware Memory Management (CAMM) - A Complete Software Approach

4.1 Background and Experimental Setup

4.1.1 NUMA architecture

Figure 2.2a shows the basic block diagram for Intel Haswell system. The nodes[1] in Haswell (similar to Westmere) processor are connected to one another through an Intel Quick Path Interconnect (QPI) [8–10]. A socket consists of a local memory which is managed by a local memory controller (MC) as shown in the Figure 2.2a, and is a Chip Multi-Processor (CMP) containing cores; all cores share a last-level cache, represented by LLC in the figure. A local memory access incurs a DRAM access delay and, if the DRAM banks are busy, an additional MC queuing delay as well. However, since the remote memory access involves moving data from the remote socket over the QPI, an additional interconnect latency is incurred. Tables 2.1 and 2.2 show the latencies in CPU cycles after a system bootup without any guest VMs running inside the ESXi for Westmere and Haswell systems, respectively. In these tables, the value in row-x and column-y represent memory access latency in CPU cycles observed from Node-x when accessing the data allocated in the

[1] We use *socket* and *node* interchangeably in the rest of this work.

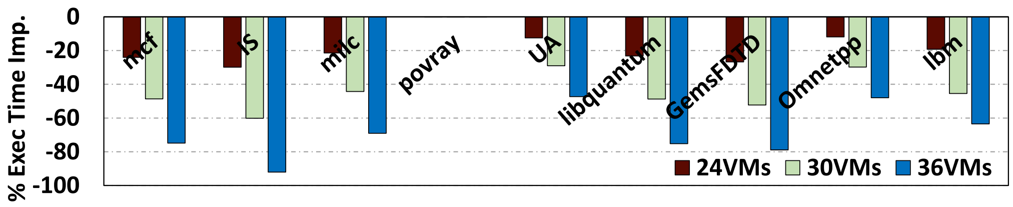


Figure 4.1: Performance degradation with varying consolidation scenarios on Intel Haswell.

local memory of Node-y. Hence, values in the diagonal (bolded) represent the local memory access latencies. From these tables, it can be observed that the local latency incurred is always lower than the remote latency.

Nodes	2
Cores/socket	18
Core Freq.	2.3GHz
LLC Size	45MB
QPI Speed	9.6GT/sec
DRAM (Capacity)	DDR4-2133 (512GB)

Table 4.1: Haswell configuration.

Nodes	8
Cores/socket	10
Core Freq.	2.27GHz
LLC Size	24MB
QPI Speed	6.4GT/sec
DRAM (Capacity)	DDR3-1333 (1TB)

Table 4.2: Westmere configuration.

4.1.2 Experimental setup

We used two Intel NUMA based systems, Haswell and Westmere to conduct our experiments. Tables 4.1 and 4.2 summarize the configurations of these machines. The Intel Haswell system shown in Table 4.1 contains 2 NUMA sockets. In each socket, there are total 36 hardware (2-way hyper-threaded) threads sharing an LLC of size 45MB. Intel Westmere system[2] shown in Table 4.2 contains 8 NUMA sockets

[2]In the interest of space, we could not present the Westmere block diagram. Please find it in slide-12 of [42].

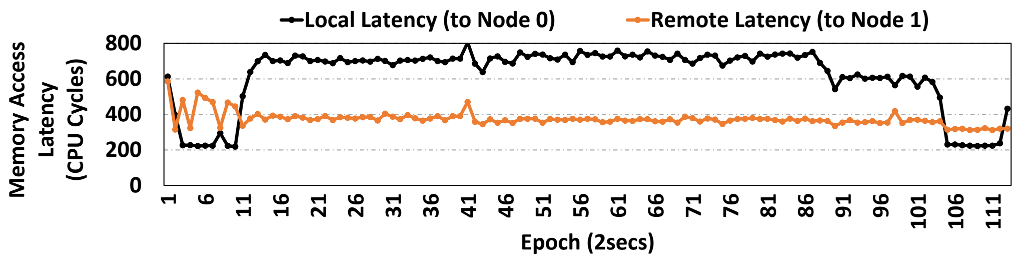


Figure 4.2: Memory access latencies noted from Node-0 in a congested environment on Haswell for npb_is workload.

and each socket contains 20 hardware threads (2-way hyper-threaded) per socket. The applications presented in Table 4.3 are run inside the guest VMs running on ESXi hypervisor. Each guest VM runs a RHEL 6.0 Operating System. For simplicity, we assume each guest VM to be running a single application, though our analysis and evaluations hold equally well for multiple applications running inside a guest VM. For our evaluations, we used multi-threaded and single-threaded applications from various benchmarks suites from HPC domain viz., NAS [43], SPEC CPU2006 suite [44], SPECOMP [45], MANTEVO [46], also the SPECJBB [47] suite and finally the graph parallel-processing multistep [48] suite. The specific applications from these suites are further categorized in to high (represented by H), medium (M) and low (L) based on their memory-intensities, measured by LLC Misses Per Kilo Instruction (LLC-MPKI) as depicted in Table 4.3.[3] Applications with LLC-MPKI greater than 15 are categorized as H, while those with LLC-MPKIs between 5-15 (inclusive) are categorized as M and the ones with LLC-MPKIs below 5 are categorized as L.

We further used homogeneous and heterogeneous workloads formed using the applications in Table 4.3 to quantify the benefits of our optimizations. The homogeneous workloads execute same applications while the heterogeneous workloads execute different applications on a given socket. To simulate real-world scenarios, where different nodes on the same machine exhibit varying degrees of congestion, in some workloads, we map high/medium/low-memory intensive applications on different nodes. However, to replicate scenarios where all the nodes experience

[3]The LLC-MPKIs reported in Table 4.3 are collected on Haswell machine by executing an instance of the same application on each hardware thread of a node. As a result, 36 threads (applications) contend for 45MB of LLC on Haswell, while on Westmere 20 threads contend for 24MB of LLC. Roughly, each threads get 1.25MB of LLC on both the machines. As a result, we observed similar LLC-MPKI values on Westmere as well.

the same degree of memory congestion, we map different instances of the same workload on all the nodes in a machine, there by covering the entire spectrum of the real-world scenarios. More details of these workloads are covered in Section 4.6.

In the rest of the work, the % Execution Time Improvement (or degradation) results reported in various figures for each workload is the geometric mean of individual applications' execution time improvement (or degradation) over the corresponding applications' baseline execution time.

Benchmark	Suite	MT ?	LLC-MPKI	Cat	Benchmark	Suite	MT ?	LLC-MPKI	Cat
mcf	SPEC2K6	N	66.9	H	GemsFDTD	SPEC2K6	N	13.33	M
npb_is	NPB	N	43.51	H	HPCCG	mantevo	N	11.81	M
npb_ua	NPB	N	42.72	H	equake	SPECOMP	Y	7.24	M
Omnetpp	SPEC2K6	N	23.01	H	specjbb	SPECJBB	Y	6.48	M
swim	SPECOMP	Y	22.162	H	CC	MULTISTEP	Y	2.45	L
lbm	SPEC2K6	N	21.62	H	mgrid	SPECOMP	Y	1.95	L
libquantum	SPEC2K6	N	19.52	H	SCC	MULTISTEP	Y	2.45	L
milc	SPEC2K6	N	18.28	H	povray	SPEC2K6	N	0.07	L

Table 4.3: Benchmarks used in our evaluation [MT: Multi-Threaded (Y - Yes, N - No), Cat: MPKI Category] .

4.2 Motivation

As can be observed from Tables 2.1 and 2.2, the access latency to local node is around 40% lower in both the systems compared to access to the neighboring remote nodes. ESXi is NUMA-aware [49] and hence tries to allocate memory from the local node, whenever possible, for better performance. However, when a high number of memory-intensive VMs are consolidated on a single node, the MC queuing and DRAM access latencies can dominate the additional interconnect latency incurred by the remote access.

4.2.1 Effect of increasing consolidation

Figure 4.1 shows how performance changes with increasing number of VMs consolidated on a socket for our homogeneous workloads on Intel Haswell system. In this experiment, each VM is pinned to a hardware thread on a processor socket. The performance degradation reported is normalized to the basecase which executes 18 VMs. As can be observed from Figure 4.1, performance degrades as the number of

VMs consolidated increases and is as high as 92% for npb_is [43] which is highly memory-intensive, while it is almost 0% for low memory-intensive applications like povray as such applications rarely access memory.

To further explain the degradation in Figure 4.1, we present the access latencies for npb_is homogeneous workload. Figure 4.2 shows the memory access latencies incurred from Node-0 to both Node-0 and Node-1 on our Intel Haswell system running homogeneous npb_is workload. As can be observed, initially when there is no congestion in the system the local node (Node-0) access latency is lower than the remote node (Node-1) access latency. However, around the 11th epoch (23 secs), the local latency to Node-0 starts increasing and becomes greater than the remote nodes' access latency indicating that the Node-0's memory bandwidth is saturated resulting in congestion. Note that, after the 11th epoch, the Node-0 latency continues to dominate the latency to Node-1 till the end of the workload execution in the 105th epoch. In such scenarios, the overall system performance would improve if the neighboring remote nodes' bandwidth could be utilized. However, to use the remote memory bandwidth effectively, one needs to answer the following questions:

1. When to allocate data in the remote node ?
2. What percentage of data needs to be allocated in the remote node for a specific topology NUMA system?
3. In which remote node (number of hops from the source node) should the data be allocated ?

These questions are important, because if data is allocated on the remote node when there is no congestion, system performance can degrade significantly as extra cycles are spent on the interconnect traversal while accessing data. Consequently, it is important to detect the congestion in the system *dynamically* so that the data can be allocated in the most appropriate node. Once the congestion is detected and the decision to allocate data in the remote node is taken, it is important to determine how much data needs to be allocated in the remote node. This decision is also important because if too less data is allocated in the remote node, system performance further degrades as the local node will still be congested. However, if a high percentage of data is allocated on the remote node, local node memory bandwidth will be under-utilized, thereby resulting in performance degradation.

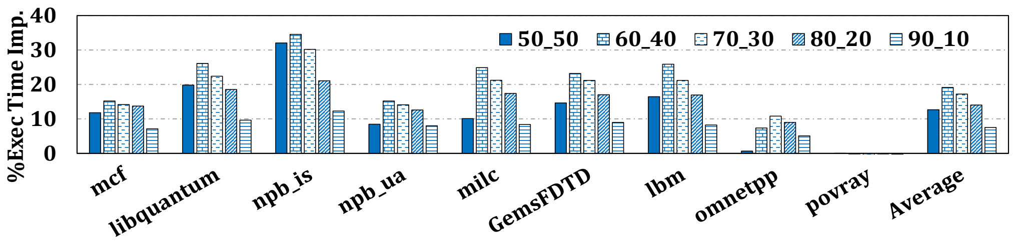


Figure 4.3: Execution time improvements with different static allocation ratios on Intel Haswell.

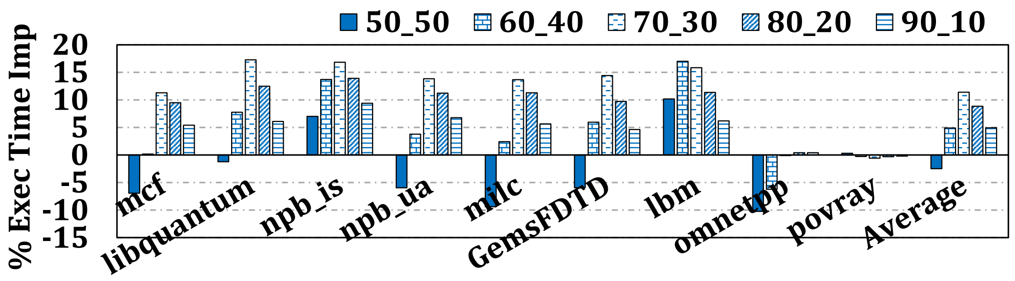


Figure 4.4: Execution time improvements with various static allocation ratios on Westmere only using 2 of 8 NUMA nodes.

4.2.2 Static allocation results

Figure 4.3 plots the percentage execution time improvement when a fixed fraction of data is allocated in the remote node on our Intel Haswell 2 NUMA node system. All the results presented are *normalized* to the basecase where all the data are allocated on the local NUMA node, Node-0. We present results for different allocation ratios, for example, the bars marked using 60_40 in Figure 4.3 represents the execution time improvement when 60% of memory is allocated on the local node, and the remaining 40% is allocated on the neighboring remote node. As can be observed, in this system, allocating 10% of data in the remote node (90_10 ratio) does not alleviate congestion and incurs considerably lower performance improvement of around 7.5% on an average. On the other hand, allocating 50% of data on the remote node alleviates congestion; however, it results in under-utilizing the local node memory bandwidth and yields a performance improvement of 12.6% on an average, which is still not very high. Allocating 60% on the local node and 40% on the remote node is optimal in this case, giving a performance improvement of 19.1% on an average. Figure 4.4 shows the performance improvement when only 2 NUMA nodes out of 8 are used in our Intel Westmere system. In these experiments, memory is allocated only on Node-0 and Node-1. Node-0 is the source node, and Node-1 is the neighboring remote destination node for VMs running on Node-0. As can be observed from Figure 4.4, 70% local node allocation and 30% remote node allocation result in the maximum performance improvement of 11.4% on an average. Similarly 70_30 yielded better performance considering 4 NUMA nodes out of 8 NUMA nodes, i.e, 30% data is spread the rest three other remote nodes. Beyond 4 NUMA nodes, latency to the remote node dominates the extra bandwidth provided by the remote node. Consequently, we do not see any performance improvement when data is allocated in the remote node beyond 4 NUMA nodes in Westmere.

Summarizing the results from Figures 4.3 and 4.4, one can conclude that memory congestion can be alleviated by using the extra memory bandwidth from the target remote nodes. There is no *one common ratio* in distributing the data across the local and remote nodes to achieve maximum performance. That is, different workloads prefer different ratios depending on the hardware configuration. Thus, we need a mechanism which can dynamically identify how congested different shared resources in a NUMA system are oblivious to the hardware configuration. These shared resources include local memory bandwidth, interconnect bandwidth, and the remote memory bandwidth. The proposed scheme should be successful in identifying congestion oblivious to the underlying hardware configuration. Hence, such a scheme should estimate congestion accurately in diversified environments employing:

- homogeneous/heterogeneous link bandwidth interconnects like AMD Bulldozer [10] and
- homogeneous/heterogeneous bandwidth memories including DRAM [50–52], AMD's HBM [3], Intel's MCDRAM [53], non-volatile memories like Intel 3D XPoint [54] and futuristic phase-change memory (PCM) [55]

4.3 Dynamic Latency Probing

Past congestion detection techniques [56–59] relied on reading the performance counters to estimate the per-socket memory intensity. However, since performance counters are *shared resources*, these mechanisms limit the number of performance counters [60] that can be used for other runtime optimizations. We implemented our dynamic probing mechanism in VMware's ESXi[4] on a real system, and observed that it gives the end-to-end information on how congested different shared resources in a NUMA system are very accurately.

Algorithm 1 Pseudo-Code for Dynamic Latency Probing.

```
1: procedure NUMA_ProbeDynLatencies(my_numa_node)
2:     for each numa_node do
3:         NUMA_ProbeDynLatency(my_numa_node, numa_node);
4:     end for
5: end procedure
6:
7: procedure NUMA_ProbeDynLatency(srcNode, tgtNode)
8:     totalAccessCycles = 0
9:     for each page in NUM_PROBED_PAGES do
10:        startCycle = RDTSC();
11:        for each probe in NUM_PROBES_PER_PAGE do
12:            /* Access the tgtNode pages (by issuing processor-loads) in a non-cacheable manner using volatile
           constructs. */
13:        end for
14:        endCycle = RDTSC();
15:        /* Let pageAccCycles represent the elapsed cycles */
16:        totalAccessCycles + = pageAccCycles;
17:    end for
18:    /* Measure and record the avgAccessLatency in the table corresponding to row srcNode and column
       tgtNode */
19:    currNodeLatArray[tgtNode] = average;
20: end procedure
```

Our dynamic latency probing mechanism is based on periodically accessing the *non-cacheable pages* in each node from every node. Algorithm 1 presents the pseudo-code for our dynamic probing mechanism. *NUMA_ProbeDynLatencies* procedure is invoked periodically on each NUMA node through timer callbacks already implemented in ESXi. From every source node, represented by *srcNode* in the pseudo-code, *NUMA_ProbeDynLatency* routine is invoked by ESXi passing the target node as an argument. To ensure only one Physical CPU (PCPU) per node probes pages from the target nodes, *NUMA_ProbeDynLatencies* callback code is only scheduled on the first PCPU of every NUMA node, except for the node where the current Timing Loop code is being executed. On this node, instead of scheduling the probing code on first PCPU, our dynamic probing latency code is executed on the same PCPU. Such an optimization prevents the pre-emption of the code currently being executed on the first PCPU for the current node. This can be observed in the pseudo-code for the timing loop presented in Algorithm 2.

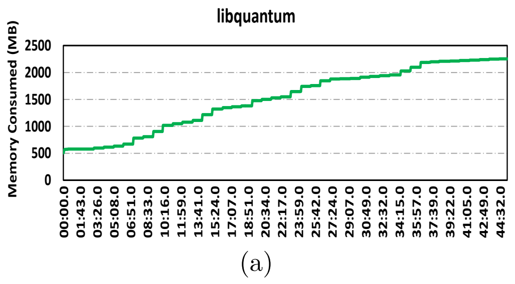
[4]We would like to emphasize again that though we implemented and evaluated the mechanism in ESXi, this mechanism is **general and can be adapted by any runtime or OS** and is **not** limited to ESXi.

This timing loop in Algorithm 2 is responsible for scheduling the dynamic latency probing code in Algorithm 1 on each node periodically.

Algorithm 2 Pseudo-Code for Dynamic Latency Probing.

```
   for each numa_node in TOTAL_NUMA_NODES do
2:     currNumaNode = NUMA_GetNumaNodeNum(MY_CPU);
       if numa_node == currNumaNode then
4:         NUMA_ProbeDynLatencies(currNumaNode);
       else
6:         fCPU=NUMA_GetFirstCPUOnNode(numa_node);
           Timer_Add(fCPU, NUMA_ProbeDynLatencies, numa_node);
8:     end if
   end for
```

As can be observed in lines 2-4 for procedure *NUMA_ProbeDynLatencies* in Algorithm 1, the for-loop code passes each node as the target node for procedure *NUMA_ProbeDynLatency*. Consequently, the source node can itself be the destination target node. In such a scenario, the PCPU running on source node accesses pages in local node.



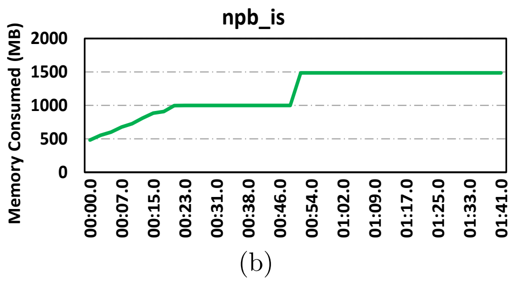


Figure 4.5: Esxtop output showing the memory allocation behavior of applications over time (in minutes).

Having understood how the dynamic probing latency code is triggered on each NUMA node, we can now look at the latency probing algorithm itself which is covered in procedure *NUMA_ProbeDynLatency* in Algorithm 1. We modified the ESXi code to allocate extra specified set of 'probe pages' on each NUMA

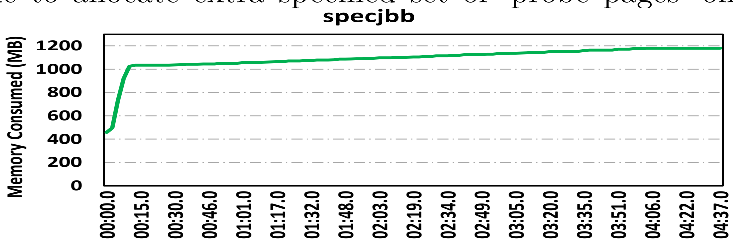


Figure 4.6: Memory allocation behavior of SPECJBB.

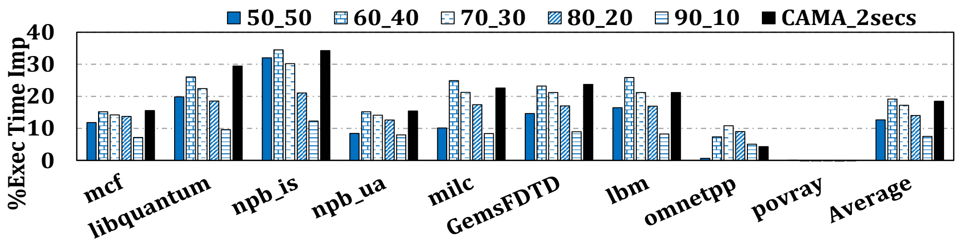


Figure 4.7: CAMA results for Intel Haswell processor.

node. Each 'probed page' is 4KB in size and is used only by ESXi for probing periodically. In the dynamic probing algorithm, these probed pages allocated per NUMA node are accessed using "volatile" construct so that the processor loads generated by accessing these probed pages do not get cached in the on-chip cache hierarchy. Hence, accesses to these probed pages always result in memory access to corresponding NUMA nodes. Since processor stores going to the memory are buffered in a separate write-queue and are drained based on the low/high watermarks set at the MC, non-cacheable stores do not truely reflect the congestion of the memory subsystem. As a result, our probing code only issues non-cacheable processor-loads to to estimate congestion. The for-loop in lines 9-19 of Algorithm 1 represents the PCPU on the source NUMA node accessing the probed pages from the target NUMA node. We use RDTSC() function to read the time-stamp counter (TSC) provided by the hardware to record the processor cycles lapsed in accessing these probed pages.

Note that the dynamic probing code itself is an additional overhead incurred by ESXi. There is a clear *trade-off* on how frequently the probing code can be triggered versus the accuracy of the memory latency information provided by probing. Probing code triggered very frequently not just pre-empts the execution of VM(s) running on the corresponding PCPUs but also causes additional memory traffic. However, if the probing code is sampled very infrequently, the memory latencies reported by the dynamic latency probing might become stale and may not be useful in alleviating memory congestion and sometimes could result in a proposed optimization to degrade the overall system performance. Two parameters in our algorithm effect the accuracy vs performance trade-off, viz, *sampling interval* and *number of memory probes.*

In the next three sections, we show how dynamic probing is used to guide memory allocation and migration strategies. Though our mechanism can be used in migrating VMs, we do *not* consider VM migration for the following reasons:

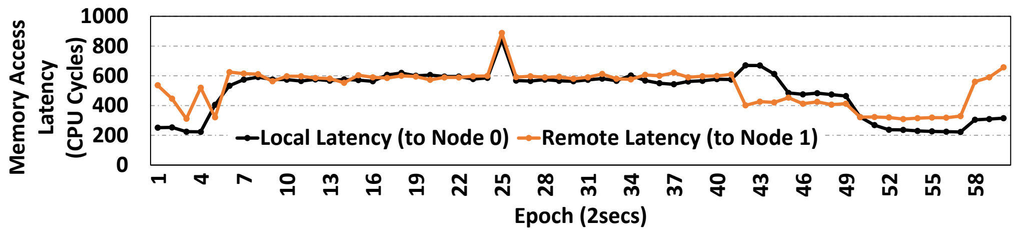


Figure 4.8: CAMA latencies for NPB IS on Haswell.

- VM migration is often an expensive operation, since migrating VM not only involves migrating the execution context, but also migrating the entire memory footprint of the corresponding VM to the remote node. Migrating the entire memory footprint is an expensive operation in terms of both energy and performance considering how quickly the memory footprint is growing in the emerging workloads [61, 62]. Migrating just the VM without it's corresponding memory results in overwhelming number of remote node memory accesses thereby degrading performance.
- VM migration from one node to another in highly consolidated environments often necessitates swapping the VMs between the source and destination nodes. This is due to the unavailability of a free PCPU that can excute the migrated VM on the target node. This VM swapping is an expensive operation as it involves not just migrating the memory, but also the cost involved in other hardware structures including: flushing the deep processor pipelines, disrupting the branch predictors, flushing TLB entries, and the locality lost in private L1D/L1I and last-level caches. And such a cost increases with increase in number of VMs to be migrated (swapped) and also by the frequency of migration.

4.4 Congestion-Aware Memory Allocation (CAMA)

Before we look into the optimizations based on the dynamic latency probing, Figures 4.5a, 4.5b and 4.6 plot the memory allocation behavior of libquantum, npb_is and specjbb applications, respectively, collected using *esxtop* utility. As can be observed, in these applications, the memory footprint grows over time and hence provides an opportunity for the memory allocator to allocate some of the data on the remote nodes dynamically when the local node is congested. In this section, we discuss Congestion-Aware Memory Allocation (CAMA), which guides allocation of data on the remote node dynamically.

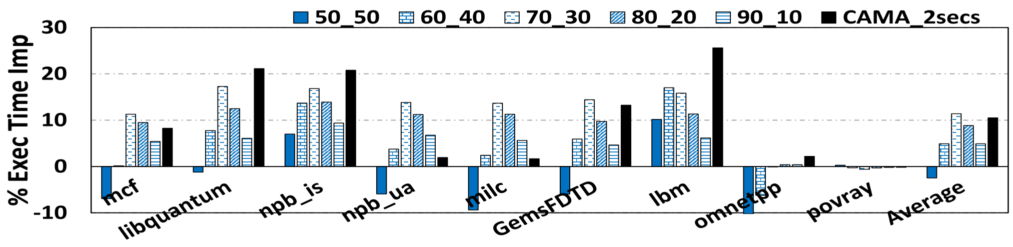


Figure 4.9: CAMA results for Intel Westmere processor.

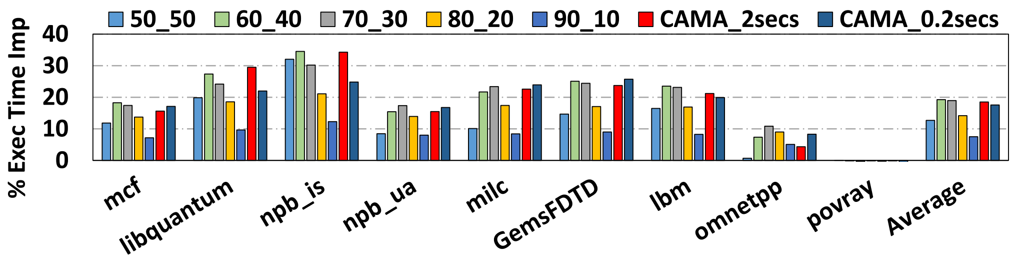
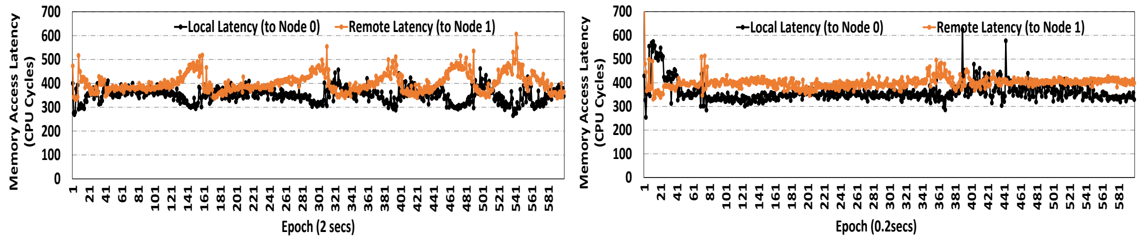


Figure 4.10: 0.2 secs epoch CAMA Improvements on Haswell.

Upon receiving a memory allocation request from a VM running on a source node, CAMA decides the target node based on the latencies incurred from the source node to the other nodes in the previous epoch. The target node chosen for allocation in CAMA is the one which incurred the lowest probed latency in the previous epoch. The intuition is that, the probed latencies recorded at the end of previous epoch gives an approximate idea of how congested different shared resources would be in the current epoch. This simple modification in the ESXi memory allocator can automatically consider the end-to-end access latencies and can account for the congestion on different shared resources. The shared resources include the local nodes' memory controllers, the inter-socket interconnect (Intel QPI/AMD HT interconnect), and the remote nodes' memory controllers. Also, this simple change in the memory allocator is flexible enough to pick the local node automatically when the access latency to the local node is lower.

In Figure 4.7, the last bar for each workload shows the performance improvement for the homogeneous workloads with CAMA. All the results presented are *normalized* to the basecase, where all the memory is allocated in the local node. The other bars representing the static allocation are shown there for comparison. The epoch duration used in these experiments is 2 secs, that is, the dynamic latencies are updated every 2 secs. As can be observed, CAMA improves the performance by 18.49% on an average and a maximum of up to 34.2% for npb_is, and also on an average CAMA is within 1% compared to the best in static allocation schemes.



(a) (b)

Figure 4.11: Dynamic latencies for VMs running mcf with epoch duration of (a) 2 secs and (b) 0.2 secs.

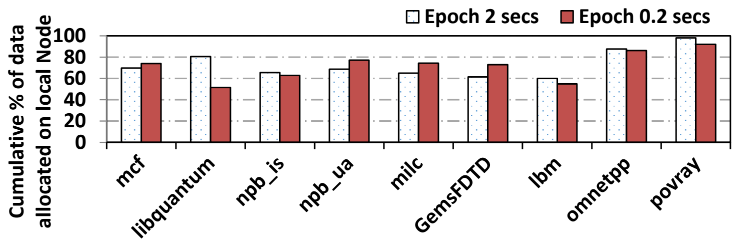


Figure 4.12: Cumulative percentage of data allocated on local node in Haswell by CAMA.

Figure 4.8 shows the dynamic latencies measured on our Haswell system from Node-0 to both Node-0 and Node-1 for CAMA in ESXi. One can see from this figure that the local and remote node latencies are almost the same at every epoch for CAMA unlike in Figure 4.2 where the local node latency is much higher than the remote node latency. Figure 4.9 shows the performance improvements in for CAMA on Intel Westmere processor; the static allocation results are reproduced for comparison. It can be observed that on an average CAMA improves the performance by 10% for Westmere. Also, from Figures 4.7, 4.9 and 4.12, comparing the static allocation and CAMA results, we note that same percentage of data allocated on local node could result in varying performance improvements.

Impact of epoch duration on CAMA: Since we use the probed latencies from the previous epoch to govern the memory allocations in the current epoch, one crucial parameter in our setup which governs the performance improvements is the epoch duration itself. Since our epoch of 2 secs translates to billions of core cycles (with a processor frequency in the order of GHz), it may be beneficial to trigger the probed latencies more frequently. Probing more frequently can give us the more up-to-date congestion information and can help the memory allocator to respond to the dynamic modulations in the application phase behavior more rapidly.

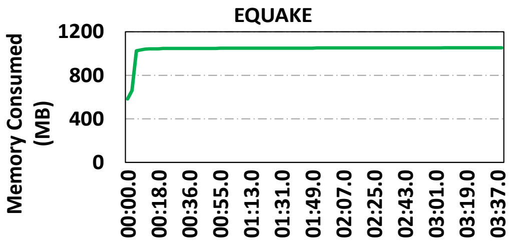
Figure 4.10 shows the performance benefits when we sample the probing code more frequently at 0.2secs epoch on the Haswell processor. For high memory-intensive applications like libquantum, npb_is, lbm we can see that the performance degrades with 0.2 secs epoch compared to 2 secs epoch. This degradation in performance is due to the fact that the probed memory accesses interfere with applications' on-demand memory accesses resulting in increased memory access latencies for the on-demand memory requests. The performance degradation in these applications is as high as 10% for npb_is compared to the case with epoch duration of 2 secs. However, applications like mcf, npb_ua, milc, GemsFDTD and omnetpp benefit from 0.2 secs epoch duration. This improvement in performance is because CAMA could adjust to subtle changes in congestion rapidly.

Figures 4.11a and 4.11b show the memory access latencies for homogeneous mcf workload for 2 secs and 0.2 secs epochs, respectively. For an epoch duration of 2 secs in Figure 4.11a, we can see the bubbles formed due to gap in latencies between the local and remote nodes around epochs 141, 301, 381, 481 and 541. This result indicates that there is still some scope for improvement when using an epoch duration of 2 secs. For epoch duration of 0.2 secs, one could see that the latencies to local and remote nodes match at every instant. This is because, all the shared resources viz., local memory bandwidth, interconnect bandwidth and remote bandwidth are utilized optimally making the end-to-end latencies to match at every instant. Hence, in workloads that benefit from smaller epoch of 0.2 secs, the performance improvement is as high as 4.5% for omnetpp, compared to an epoch duration of 2 secs. On an average, an epoch duration of 0.2 secs yields 17.5% increase in performance compared to the basecase and the corresponding performance improvement for an epoch duration of 2 secs is 18.4%. Figure 4.12 gives the cumulative percentage of data allocated on the local node for epochs of 2 secs and 0.2 secs using CAMA. For both the epochs, high memory-intensive applications like mcf and npb_is have more than 30% of their data in the remote node.

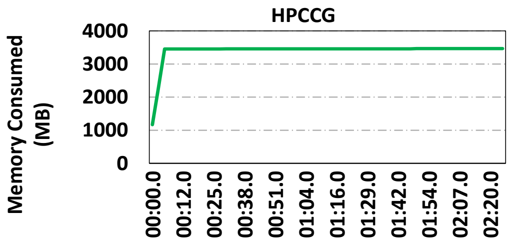
4.5 Congestion-Aware Page Migration (CAPM)

CAMA is ineffective in the following scenarios:

1. Memory system is congested during data allocation, but it is no longer congested



(a)



(b)

Figure 4.13: Esxtop output showing memory allocation behavior of the applications over time (in minutes).

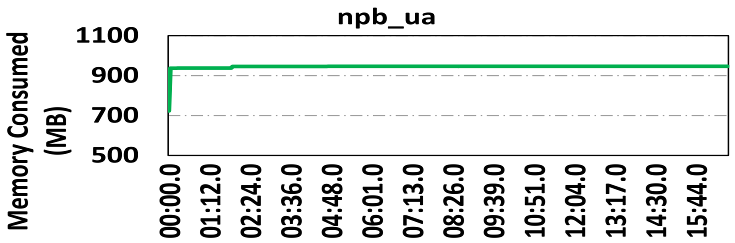


Figure 4.14: Memory Allocation behavior of NPB_UA.

when the data is being accessed.

2. Memory subsystem is not congested during data allocation, however, it gets congested during execution of workload.

Figures 4.13a, 4.13b and 4.14, show the memory allocation behavior of different classes of applications over time. As can be observed, all of the data is allocated in the first few seconds of the program execution. In such scenarios, as local node memory will not be congested initially, CAMA allocates all the data in the local node it cannot alleviate congestion.

Therefore, we need a dynamic mechanism which can fix the incorrect decisions made by CAMA at runtime or augment CAMA to reap maximum overall system performance. Motivated by this, we propose our second optimization, Congestion-Aware Page Migration (CAPM) which dynamically *migrates* pages across nodes based on the probed latencies described in Section 4.3. In this section, we elaborate on the intricacies in designing CAPM in isolation in the absence of our first proposed optimization CAMA. From Figures 4.8 and 4.11b, in presence of congestion, it can be observed that maximum overall performance is obtained when the shared resources are utilized optimally, i.e, when the gap between end-to-end memory latencies of the local node and the remote node is minimum in every epoch. Hence, based on dynamic probed latency in the previous epoch, CAPM tries to migrate pages such that the latencies to the local node and remote node(s) match in every epoch. When local node becomes congested, CAPM migrates the data from local node to the remote node, but when the latency to the remote node increases later, data is migrated back from remote node to local node in next epoch to balance end-to-end latencies. The key factors that effect improvements in CAPM are:

- Candidate pages chosen to migrate from source to destination nodes.
- Number of such candidate pages to be migrated per epoch, referred to as *migration rate.*

Algorithm 3 Pseudo Code for setting the migration rate in CAPM.

```
1: currNumaNode = NUMA_GetNumaNodeNumFromCPU(MY_CPU);
2: currNodeLatencyArray = NUMA_GetDynLatCyclesArray(currentNumaNode);
3:
4: currNodeLatencyArray contains the dynamic probed latencies from the current NUMA Node to all the other
   NUMA nodes.
5:
6: Iterate currNodeLatencyArray and find the node which incurs minimum latency from the current NUMA node
   and the latency value. Let minLatencyNode and minNodeLatency represent the corresponding values. Also,
   calculate the average of the latencies. Let avgLatency represent the average value.
7:
8: if currNumaNode ! = minLatencyNode then
9:     /* Local NUMA node is congested, pages are to be migrated from local current NUMA node to the
   neighboring remote NUMA node. */
10:     latency_gap = currNodeLatencyArray[currNumaNode] - avgLatency;
11:     if latency_gap > ($\eta_{thresh} * avgLatency$) then
12:         Set migration rate here
13:     else
14:         $migrationrate = 0$;
15:     end if
16: else
17:     /* Neighboring remote NUMA node is congested, pages are to be migrated from neighboring remote
   NUMA node to local current NUMA node. */
18:     latency_gap = currNodeLatencyArray[neighboringNumaNode] - avgLatency;
19:     if latency_gap > ($\eta_{thresh} * avgLatency$) then
20:         Set migration rate here
21:     else
22:         $migrationrate = 0$;
23:     end if
24: end if
```

Candidate Pages for Migration: Since programs exhibit locality [63] during execution, not all the pages allocated are accessed in all the epochs. In VMware ESXi and traditional OS's like Linux, such pages accessed can be identified by poisoning the page table entry (PTE) by setting its reserved bit similar to [63]. Such a poisoned page is flushed from the TLB and a corresponding processor load/store will incur a TLB miss. Upon a miss, a hardware pagetable walk is triggered following which a BadgerTrap routine [64] is executed accounting for the page access. BadgerTrap handler further resets (unpoisons) the reserved bit

Algorithm 4 Pseudo Code for setting CAPM migration rate.

```
1: currNode = NUMA_GetNumaNodeNumFromCPU(MY_CPU);
2: currNodeLatArray = NUMA_GetDynLatCyclesArray(currNode);
3: /* currNodeLatArray –> Latencies from the current Node to all the other nodes. minLatencyNode –> Node
   incurring minimum latency from current Node. */
4: if currNode ! = minLatencyNode then
5:     /* Local node is congested, pages are to be migrated from local current node to the neighboring remote
   node. */
6:     lat_gap = currNodeLatArray[currNode] - avgLatency;
7:     if (lat_gap > ($\eta_{thresh} * avgLatency$))&&($freespace\_in\_remote\_node$) then
8:         Set migration rate here
9:     else
10:        migrationrate = 0;
11:    end if
12: else
13:    /* Neighboring remote NUMA node is congested, pages are to be migrated from neighboring remote
   NUMA node to local current NUMA node. */
14:    lat_gap = currNodeLatArray[neighboringNode] - avgLatency;
15:    if (lat_gap > ($\eta_{thresh} * avgLatency$))&&($freespace\_in\_local\_node$) then
16:        Set migration rate here
17:    else
18:        migrationrate = 0;
19:    end if
20: end if
```

in PTE and caches the translation in TLB and later re-poisons the PTE. Hence the number of badgertraps accounted for each page can be used to distinguish an accessed page from a non-accessed page in a given epoch. From our offline-analysis of each application, we identified that in every epoch, total memory footprint of the accessed pages is in the order of several Mega Bytes (MBs) which span over few thousands of 4KB pages. In CAPM, the candidate pages to be migrated are only chosen from the accessed set of pages. Hence the migration rates used in CAPM will be in the order of few thousands per epoch to alleviate congestion.

Target Migration Rate per epoch: Migrating a candidate page from source node to destination node incurs following steps: (a) allocating a new page in the destination node, (b) copying the entire page contents from the source node memory to the target node memory, (c) shooting down the cached translations inside the TLB and (d) updating the page table entry. Prior work [65] accounted for the TLB shootdown overheads to be as high as 11000 CPU cycles for 16 threads. Considering the above overheads, the target migration rate per epoch, plays a significant role in the overall system performance. Since our CAPM supports migration in both the directions (local to remote and remote to local), the migration rate should be chosen such that the overheads are minimal. A lower migration rate migrates fewer pages per epoch, and hence it will take more time to alleviate congestion, resulting in low improvement in performance. On the other hand, a high migration rate causes more pages to migrate per epoch, thereby alleviating the congestion faster.

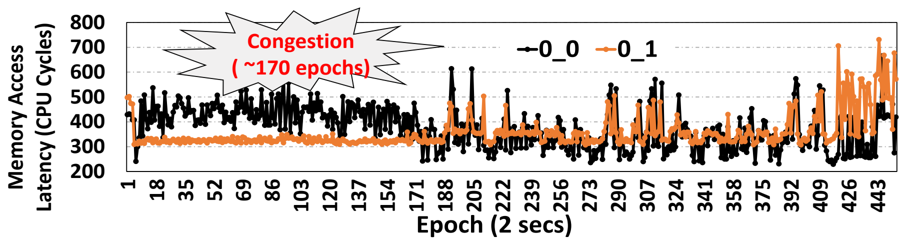


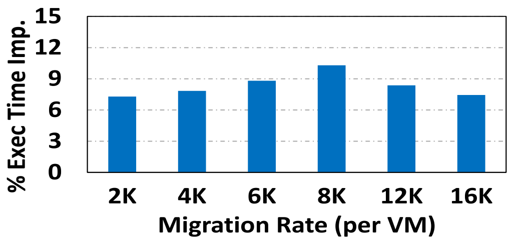
Figure 4.15: Dynamic latencies without any migration.

However, at the end of the epoch, due to migration, the latency to the neighbouring NUMA node will become higher than that to the local NUMA node. As a result, pages are now migrated from the neighbouring node to the local node in the next epoch, thereby causing the pages to migrate back-and-forth between the local and the neighboring remote nodes, causing a *ping-pong effect*. Clearly, such a ping-pong effect is not desirable warranting the migration rate to be chosen carefully.

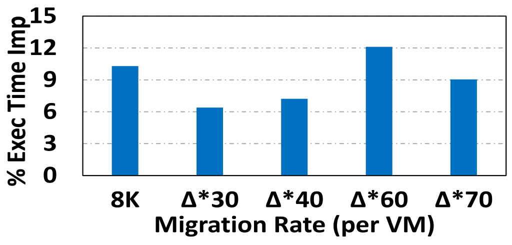
When the local node is congested, it is clear from Figure 4.8 that, the maximum performance is achieved when the measured end-to-end latencies between local and neighboring nodes are close to each other in every epoch. Hence, the desired probed latency at every epoch to both the local and remote nodes should be the average of the observed latencies. The pseudo-code for setting the migration rate in CAPM is presented in Algorithm 4. At the end of every epoch, for every node, the algorithm involves identifying the congested node by comparing it's local NUMA node latency with the neighboring remote node. Once a congested node is identified, the lat_gap, defined as difference in the latency to the congested node and the average expected latency is computed, as shown in lines 10 and 18. If the lat_gap exceeds the expected average by a threshold, denoted by η_{thresh}, CAPM migrates pages from the congested node to the neighboring node[5], else not. If η_{thresh} is too low, less latency-gap can be tolerated and thus too many pages could be migrated which could possibly trigger the ping-pong effect. If η_{thresh} is too high, too much latency gap will be tolerated and consequently not enough pages are migrated, resulting in low overall performance. Based on our experimentation on all our workloads, we identified 5% to be a reasonable value for η_{thresh}(used as threshold in this work).

To explain various results in CAPM, we use the following heterogeneous workload: equake(2), povray(2), mgrid(2), mcf(2), swim(4). This heterogeneous work-

[5]Please note from lines 7 and 15 of Algorithm 4, migration rate will be 0 if there is no free space available in the destination node. Hence, CAPM tries to alleviate congestion opportunistically as long as there is freespace available in the destination node.



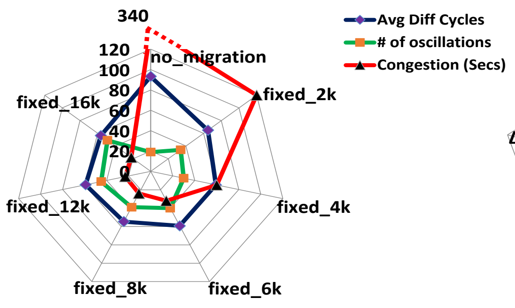
(a)



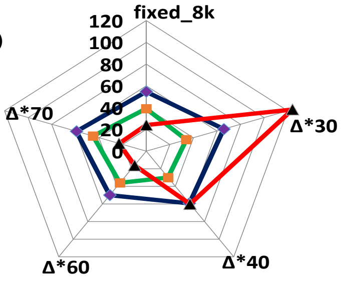
(b)

Figure 4.16: (a) Static, (b) Dynamic migration rate results.

load contains multi-programmed and multi-threaded applications with varied memory intensities. Figure 4.15 represents the dynamic latencies measured in the basecase where all the data is allocated on the local node. It can be observed that the local node is congested for around 170 epochs (340 secs) since the latency to the local node is higher compared to that to the remote node.



(a)



(b)

Figure 4.17: (a) Static and (b) Dynamic migration rate results.

Once CAPM decides to migrate candidate pages based on the latency gap and η_{thresh}, the next important question is how many candidate pages to migrate per VM per epoch to get the best overall system performance. Migration rates per VM can be set per epoch either statically or dynamically. As covered in " Candidate Pages for Migration" discussion, since the working set sizes of the accessed pages are in the order of several MBs, few thousands of candidate pages need to be migrated per epoch to alleviate congestion. Figure 4.16a shows the performance improvements for the heterogeneous workloads when migration rate is varied from 2000 to 16000 candidate pages per VM. As the migration rate is increased from 2000 to 8000, performance improvement over the baseline increases from 7.2% to 10.3% and beyond 8000, the performance degrades from 10.3% to 7.45% as the migration rate is further increased from 8000 till 16000. Hence, 8000 pages per epoch per VM is the best migration rate.

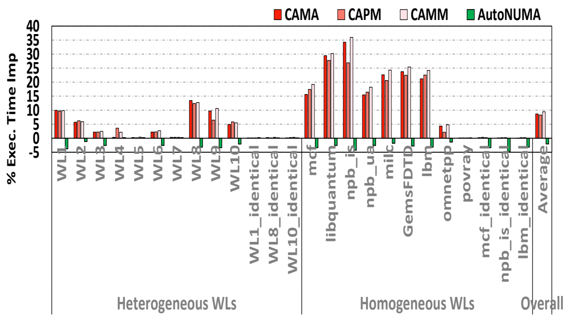
This variation can be quantified using following parameters:

1. Congestion duration in seconds,
2. Number of oscillations in the probed latencies, and
3. Average difference between probed latencies per epoch.

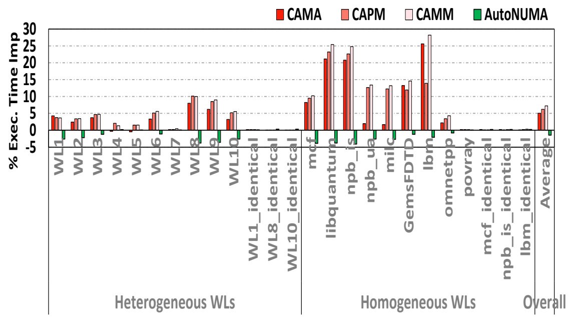
The congestion duration is defined as the time in seconds during which the probed latency to the local node is higher than that to the remote node over the entire execution. From Figures 4.15 and 4.17a, the congestion duration is around 340 secs for the basecase. Smaller congestion duration indicates that a specific migration rate could mitigate the congestion faster. As the migration rate is changed from 2000 to 16000, the congestion duration changes from 120 secs to 22 secs.

The number of oscillations in probed latencies represents the ping-pong effect in observed latencies due to back-and-forth migrations of pages between NUMA nodes. With its value initialized to zero, once the migration is triggered due to congestion, the value is incremented every time the probed latencies to local and remote nodes cross each other. To get an understanding on how its value is calculated, for example, for the basecase in Figure 4.15, initially as there is no congestion, its value remains zero for few epochs as the local node is not congested. As the local node gets congested over time, the first epoch when the latency to local node crosses the remote node latency, its value is incremented to 1 and the value keeps incrementing every time the latencies cross each other. As can be observed in Figure 4.17a, the number of oscillations for basecase is 19, which is the lowest among the other values for different static migration rates as the data is not migrated in the basecase. The only oscillations that happen in latencies in basecase is because of the variations in application behavior triggering changes in the overall memory intensity of the socket. As the static migration rate increases from 2000 to 16000, the number of oscillations increases from 19 to 49. With the higher migration rate, the pages keep migrating back-and-forth between the local and remote nodes, thereby resulting in higher number of oscillations. For a migration rate of 8000, the number of oscillations suffered is 39.

The average difference in CPU cycles between the probed latencies per epoch is defined as the average of absolute difference in probed latencies between the local node and remote nodes at every epoch. Figure 4.17a shows how the average



(a)



(b)

Figure 4.18: Overall execution time improvements for (a) Intel Haswell and (b) Intel Westmere.

difference in CPU cycles is effected by the static migration rate. For the basecase, as the data is not migrated, the local node is congested, and hence, the absolute difference in probed latencies per epoch is as high as 93 CPU cycles. As the migration rate is varied from 2000 to 16000, average value changes from 65 cycles to 56 cycles with a minimum value of 54 cycles for a migration rate of 8000.

The maximum performance improvements occur when all the above mentioned parameters experience lower values, which seems to be the case for a migration rate of 8000. Though the static migration rates can improve performance, being agnostic to the gap in probed latencies might not yield the maximum performance. Observing this, we propose using the current absolute latency gap at the end of the epoch, represented by Δ_{cycles}, to calculate the dynamic migration rates so that we migrate more pages if the latency gap is high and fewer pages if the latency gap is not too high. Figure 4.16b shows the improvement in performance for different dynamic migration rates, and the corresponding values for the three parameters discussed above are plotted in Figure 4.17b (results for static migration rate 8000 is just shown for comparison). As can be observed, ($\Delta_{cycles} \times 60$) yields maximum system performance with 12.1% in overall system performance[6]. Hence, CAPM is successful in alleviating congestion in scenarios where CAMA is ineffective.

[6]In all our workloads, we observed that this heuristic yields maximum performance.

4.6 Congestion-Aware Memory Management (CAMM)

In this section, we discuss and evaluate how CAMA and CAPM interact with each other. We refer to this combined scheme, which includes both CAMA and CAPM, as Congestion-Aware Memory Management (CAMM). CAPM employed a dynamic migration rate of $\Delta_{cycles} \times 60$ and the epoch duration used is 2 secs. To give an overall picture of how the overall system performance is impacted by our schemes, we present the full evaluation results for CAMA, CAPM and CAMM using homogeneous and heterogeneous workloads. Tables 4.4 and 4.5 show heterogeneous workloads for Westmere and Haswell respectively. Apart from WL1 to WL10 heterogeneous workloads, we have three additional workloads, WL1_identical, WL8_identical, WL10_identical where all the nodes in a machine execute the same workloads thereby causing equal amount of congestion to their corresponding local memories. Similarly we use mcf_identical, npb_is_identical and lbm_identical homogeneous workloads that result in same amount of congestion in all the nodes of a machine. In the heterogeneous workloads, since some applications finish execution earlier, a node may not be congested allthrough the workload execution which might not depict some real-world scenarios. To cover such scenarios, we restart applications which finish earlier till the point where each application in the workload finishes execution at least once.

	Workloads	**MPKI Category**	**Restart Enabled?**
WL-1	mcf(2), libquantum(3), is(3), milc(2), lbm(2), swim(1)	H	No
WL-2	hpccg(2), milc(1), is(1), libquantum(1), equake(1), specjbb(1), lbm(1)	H+M	No
WL-3	GemsFDTD(4), specjbb(1), equake(1), hpccg(2)	M	No
WL-4	hpccg, povray(2), cc(1), equake(1), mgrid(1), GemsFDTD(1)	M+L	No
WL-5	povray(4), scc(2), mgrid(2)	L	No
WL-6	povray(1), mcf(2), libquantum(2), cc(1), is(1), ua(1)	H + L	No
WL-7	GemsFDTD(1), mcf(1), povray(2), lbm(1), milc(1), specjbb(1), libquantum(1), ua(1), swim(1), mgrid(1)	H + M + L	No
WL-8	mcf(2), libquantum(3), is(3), milc(2), lbm(2), swim(1)	H	Yes
WL-9	hpccg(2), milc(1), is(1), libquantum(1), equake(1), specjbb(1), lbm(1)	H + M	Yes
WL-10	povray(1), mcf(2), libquantum(2), cc(1), is(1), ua(1)	H+ L	Yes

Table 4.4: Heterogeneous workloads for Intel Westmere.

Overall CAMM results: In Haswell, CAMM improves the overall performance on an averge by 9.5%, while CAMA and CAPM imrove the performance by 8.6% and 8.2% respectively. Similarly, on Westmere, CAMM improved the performance on an average by 7.2%, while CAMA and CAPM improved the performance by 5.1%

	Workloads	**MPKI Category**	**Restart Enabled?**
WL-1	is(6), ua(4), mcf(4), libquantum(5), milc(4), swim(2), omnetpp(2)	H	No
WL-2	GemsFDTD(3), lbm(3), libquantum(2), mcf(3), omnetpp(3), is(4), ua(1), equake(2), swim(1)	M+ H	No
WL-3	GemsFDTD(4), hpccg(3), equake(2), specjbb(3)	M	No
WL-4	equake(3), mgrid(2), GemsFDTD(4), specjbb(1), hpccg(1)	M + L	No
WL-5	povray(8), cc(3), scc(3), mgrid(1)	L	No
WL-6	lbm(2), libquantum(3), mcf(2), milc(3), omnetpp(2),cc(2), scc, is(2), npb_ua(2), mgrid(1), swim(1),	H+L	No
WL-7	GemsFDTD(2), lbm(2), mcf(2), milc(1), omnetpp(2), hpccg(1), cc(1), scc(2), is(1), ua(1), equake(1), swim(1)	H+M+L	No
WL-8	is(6), ua(4), mcf(4), libquantum(5), milc(4), swim(2), omnetpp(2)	H	Yes
WL-9	GemsFDTD(3), lbm(3), libquantum(2), mcf(3), omnetpp(3), is(4), ua(1), equake(2), swim(1)	H+M	Yes
WL-10	lbm(2), libquantum(3), mcf(2), ua(2), mgrid(1) milc(3), omnetpp(2), cc(2), scc, is(2),	H+L	Yes

Table 4.5: Heterogeneous workloads for Intel Haswell.

and 6.2% over the basecase, respectively. Since all the nodes are equally congested in the heterogeneous WL1_identical, WL8_identical, WL10_identical workloads and homogeneous mcf_identical, npb_is_identical, lbm_identical workloads, the CAMM does not allocate/migrate any data to the remote node. This is because the remote node probed latencies are greater than the local node probed latencies in every epoch owing to the additional interconnect traversal latency thereby resulting in 100% local accesses. As a result, there is no improvement in performance in these workloads over the baseline as can be observed in Figures 4.18a and 4.18b.

Homogeneous-vs-Heterogeneous WL Results: From figures 4.18a and 4.18b it can be observed that homogeneous workloads yield better improvements compared to the heterogeneous ones. This is because, homogeneous workloads have high overall memory intensity compared to the heterogeneous ones. For example, comparing workloads WL8 and npb_is, which are highly memory-intensive in their respective (heterogeneous/homogeneous) categories, the probed latencies look as follows: For WL8: Node0→Node0: 434 cycles; Node0→Node1: 340 cycles; Delta: 94 cycles. For npb_is: Node0→Node0: 630 cycles; Node0→Node1: 375 cycles; Delta: 255 cycles. The homogeneous npb_is workload is around $2.7x$ more congested compared to heterogenous WL8, resulting in larger improvements for npb_is. This variation in memory intensities between homogeneous and heterogeneous workloads is primarily due to application “phase behavior". In a homogeneous workload since the same applications are run on all the cores, at any given moment all the applications are in the same phase, a node is either highly congested or not. For heterogeneous-workloads, since different applications are running concurrently, they are in different phases at any given epoch, thereby causing lower congestion relatively.

Haswell-vs-Westmere Improvements: From Figures 4.18a and 4.18b, it can be observed that the improvements on Haswell are relatively higher compared to the

Westmere processor. This difference is because Haswell system runs 36 VMs per socket, while Westmere only runs 20VMs per socket. Hence, in our experiments, we observed that Haswell system is more congested overall compared to the Westmere system. Since our mechanism is successful in alleviating congestion, CAMM yields better improvements in Haswell compared to Westmere.

Dynamic Probing Overhead Analysis: In our dynamic probing code presented in Algorithm 1, NUM_PROBED_PAGES and NUM_PROBES_PER_PAGE play an important role in the overall performance improvement. Too few requests might not give the accurate congestion information, and too many probe-requests will interfere with on-demand requests issued by VMs. Based on experimentation with all our workloads, we determined NUM_PROBED_PAGES and NUM_PROBES_PER_PAGE with values 20 (per-node) and 8 respectively, could capture the end-to-end congestion information accurately. For this configuration, our probing-code accesses 160 cache lines from a node to determine the average end-to-end access latency for every 2 secs epoch. The npb_is (homogeneous) workload (which is the most memory-intensive workload among our workloads) running on Haswell, experienced an average memory access latency of 630 CPU cycles (per memory request) to the congested node (Node-0) in the baseline. In such scenarios, our default probing overhead is 48 μsecs for every 2 secs probing. However, with our proposed CAMM optimizations, average memory access latency reduces significantly, further reducing probing overheads to few μsecs, which is significantly better over prior proposals [58].

Comparison with AutoNUMA: The current version of Linux is NUMA-aware and supports AutoNUMA [66] feature to minimize remote node accesses. AutoNUMA migrates threads/memory to co-locate threads and data to minimize remote node accesses. It tracks local-vs-remote faults by invalidating few TLB entries, generating page-faults when those pages are accessed. However, unlike CAMM, autoNUMA does not migrate memory pages anymore if a workload incurs 100% local accesses irrespective of the congestion. As can be observed in Figures 4.18a and 4.18b (green-bar), congestion-agnostic autoNUMA implemented in ESXi infact degraded the performance over the baseline on an average by 2.1% and 1.4% on Haswell and Westmere respectively, as the additional time is spent servicing the artificially induced page-faults.

Chapter 5

Near-Data Computing in Manycores

5.1 Introduction

As we step into the exascale computing era, the performance and power/energy costs of data movement are expected to be orders of magnitudes higher relative to the corresponding computation costs [67]. In other words, most of the execution cycles and power required to execute a large-scale application is expected to be spent in moving data across different system components. Unfortunately, existing hardware and software based data optimization techniques are not well-positioned to address this growing data movement problem, mainly because "data" in those techniques is taken into account as a "side effect" of performing computations, *not* as a first-class optimization target.

This observation has motivated for a shift of focus from *computation-centric systems*, which have dominated the parallelism landscape so far, to *data-centric systems,* where data movement across different system components is treated as a *first-class optimization metric.* The prime example of this shift is the emerging *in-situ* computing paradigm [68] in supercomputing, in which computation is migrated to where data are, instead of the other way around. While several recent works [69–71] have investigated the mechanisms using which in-situ computing can be enabled, their main focus is on large-scale cluster-based systems or supercomputers.

It is to be noted that in-situ computing is an example of the *near-data computing* paradigm. With the emergence of manycore systems (e.g., Intel Xeon Phi with 60+ cores [72] and Tilera TILE-Gx with 9-72 cores [73]) where a large number of cores are connected to one another using an on-chip network, the cost of data movement is increasingly becoming a concern at a node level as well. For example, moving data from one corner of a node (a manycore architecture) to another can take a significant number of cycles [74] and large amounts of energy [75].

Similarly, making an off-chip memory access can be tens to hundreds of times more costly than making a first-level cache access. As a result, we believe that *near-data computing can be a viable paradigm in the context of a manycore system as well.* To our knowledge, there is *no* prior work that uses near-data computing targeting emerging manycore systems.

One of the critical questions to address when considering near data computing in the context of emerging manycores is its potential and limitations. That is, one would be very much interested in an early assessment of the potential of near-data computing in improving performance and reducing power/energy consumption. Also, the prior near-data computing (NDC) research proposed in the past [76–78] proposed off-loading computations to the stacked/off-chip DRAMs. The major challenges in designing a NDC system using these prior NDC architectures as discussed in [76] include: (1) the support for virtual memory and (2) data coherency, making the overall system design more challenging. Using a set of multi-programmed and multi-threaded workloads and focusing only on the performance aspect, this work presents a detailed evaluation of the potential benefits and limitations of near-data computing in an on-chip network-based manycore. The main **contributions** of our work include:

• We conduct a "limit study" for near-data computing in manycores. Specifically, we evaluate the potential of three alternative incarnations of near-data computing. The first of these, referred to as NDC-1 in this work, brings the computation to data, under a given data placement to memory. The second strategy, called NDC-2, co-locates computation and data within the multicore chip, and finally, the third strategy, referred to as NDC-3, goes one step further and takes computation to DRAM. A common characteristic of these strategies is that they work under the assumption of zero-cost data movement and infinite resources to offload the computations to. Our extensive experiments with these strategies indicate that NDC-1, NDC-2 and NDC-3 can improve application performance by 31%, 66% and 75%, respectively, on average, evaluated with various multi-threaded and multi-programmed workloads.

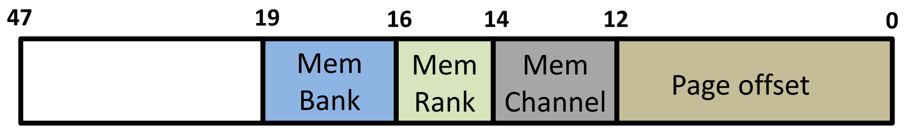


Figure 5.1: Memory address.

• We discuss what type of architectural/system support is needed to approximate near-data computing in multicores. Specifically, we present experimental results with configurations where the assumptions we made in the previous step regarding cost-free movement and infinite offload resources are relaxed. The results with these more realistic configurations indicate that, with modest hardware support, such as express virtual channels, near-data computing can bring great improvements (48% on average, in application performance).

5.2 Background

5.2.1 Network-On-Chip (NoC) Basics

In an on-chip network, each node has a router associated with it. Consider the example shown in Figure 2.1, where a data access by the core causes a miss in the L2 cache. The request message passes through 4 routers on the way to its destination L3. It is to be noted that, in x-y routing, the message first traverses the network horizontally (along the x-axis) before going vertically (along y). Each message is subdivided into flits (header flit and tail flits). These flits travel through each router. On receiving a flit, a router first decodes it, buffers it at one of its multiple input buffers, decodes its next router, and forwards it to the next router.

5.2.2 Memory Request Routing

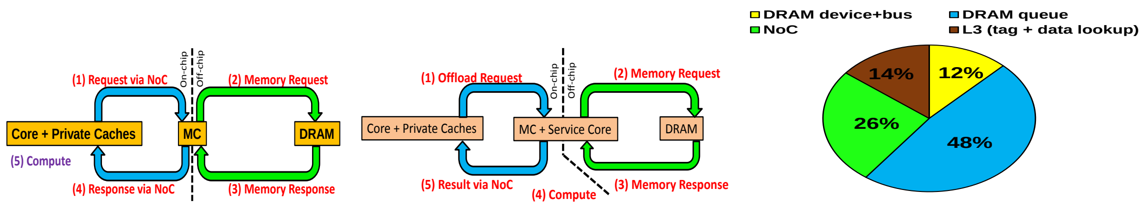
When an application requests memory, the operating system (OS), which maintains a free-list of memory blocks, allocates some portion of the physical memory in the DRAM and then creates a corresponding virtual address to physical address translation in the page table. At the hardware level, this physical memory location is mapped to a memory controller based on the static address mapping. Memory locations are generally interleaved across memory controllers at a page-level which is usually 4KB or 8KB. For example, if a page is interleaved at a 8KB granularity, the first 8KB of memory goes to the first controller, the next 8KB goes to the next memory controller, and so on. Figure 5.1 shows the parts of a memory address for static address mapping at 8KB granularity. The first 13 bits in a memory address represent the page-offset in a 8KB page. Assuming there are 4 memory channels in the system, the next 2 bits represent the memory controller in which this memory address resides. The next other bits to the left determines the other sub-components like ranks/banks and sub-arrays. Hence, based on the interleaving employed by the hardware, an access to a memory location can ends up in an access to a corresponding memory controller on the chip.

5.2.3 Anatomy of a Memory Request

Once a core issues a LOAD or STORE request, first the address is looked up in the local L1, then the L2. If the requested address is not present in the local private caches (L1 and L2), a read command is sent to the L3 bank corresponding to the address through the NoC. If the data is not present there either, an off-chip request is forwarded to the MC that holds the address. The target MC for each memory request depends on the OS address mapping. Figure 2.1 above shows this pictorially. Note that the access latency is a function of multiple parameters. The first is the distance between the core and the MC that serves the request. The longer the distance between the core and MC, the higher the chances for the packet to face network contention. The second factor in the access latency is the on-chip network traffic. As the network is a shared resource that serves a lot of requests and correspondingly a lot of responses, from L2 to L3, L3 to MC, and so on. Thus, packets can be significantly delayed in the network because of contention from other packets. The third factor contributing to access latency is MC queuing which could inturn be effected by the DRAM refreshes [79] and finally the DRAM service latency. This partly is determined by the row-buffer management (open or closed page) and scheduling policies (e.g. first-come first-served, FR-FCFS [80], etc.) employed at the MC.

5.2.4 Structure of our Target Applications

In this study, we target scientific applications from the high-performance computing domain. These applications typically process extremely large datasets in an iterative fashion. Their main kernels tend to repeat the same computation over these large



(a) (b) (c)

Figure 5.2: High-level view of (a) the conventional computing model, (b) our approach, and (c) Average uncore latency breakdown (for all workloads used in our evaluation) of memory requests, including L2 to L3 accesses, L3 to memory, and memory latency. The NoC latency includes transmission, processing and queuing latencies.

data sets, for example to model a system over time or space, and as a result these applications have high levels of data parallelism. In our chosen applications, the kernels contain data-intensive phases and not computationally intensive phases. These data-intensive phases of applications demand a lot of memory bandwidth rather than compute power.

5.3 High-Level View of Our Approach and Execution Model

In our target manycore architecture, we evaluate *multi-programmed* as well as *multi-threaded* benchmarks. The applications whose computations are offloaded closer to or at the DRAM are the referred to as *target applications*, while the other applications running at the same time are referred to as the *co-runner applications*. The computations for the co-runner applications are *not* offloaded and the data required for the computation are pulled to the corresponding cores. The data from the co-runner applications travel on-chip through various levels of memory hierarchy including private L1 and L2 caches, and the shared SNUCA L3 cache. As a result, these co-runner applications contribute to the interference for the shared resources on-chip. Note that both the target and co-runner applications can be single-threaded or multithreaded. In this work, we chose five common kernels as our target applications that are frequently used in scientific computing: *stencil*, *sparse matrix-vector multiplication* (SPMV), *scan*, *histo*, and *sum of absolute differences* (SAD). These target kernels are evaluated for single-threaded as well as multi-threaded executions. We have also evaluated multiple target applications running with other co-runner applications on the same manycore processor. The co-runner applications chosen have varying memory footprints and hence they have various on-chip network and memory bandwidth requirements, and put different degrees of pressure on the target applications.

Figure 5.2(b) shows a high-level view of our approach compared to the conventional model of computing on a manycore. In the conventional model, data are moved from the DRAM into the on-chip caches before being operated on. As shown in Figure 5.2(a), the core generates a cache miss that (1) travels through the NoC to the memory controller, and (2) is sent to the memory. After the requested data is fetched from the memory, it must be sent (3) back to the MC and (4) back to the core via the NoC. Finally, once the core has all the data it needs, it can complete the computation (5).

In contrast to the conventional model, in our near-data computing-based approach, the computation is assumed to take place at *service cores* located on-chip next to the memory controllers[1]. As shown in Figure 5.2(b), the application first indicates which computation is to be offloaded (1) to the MC. The MC (2) requests and (3) receives the necessary data in the same way as the conventional approach and provides it to the service core. Then, the computation (4) can be performed close to the memory controller. Finally, the result of the computation is sent back (5) to the application core. Depending on the application, this result data can be much smaller than the original input data. Note that, depending on how the data are distributed across the memory channels, parts of the computation may be mapped to different memory controllers. In this case, the result of the computation at the service core may be an intermediate result that will be combined with other results once it reaches the application core. This is similar to the combination that occurs after the join in fork-join parallelism. The application core itself may also be using a similar parallel programming model with the intermediate results being locally cached.

5.3.1 Cost of Data Movement

NoC queuing latency is one of the bottlenecks for memory requests in the system [81], and it increases both by the distance (number of hops) a message must travel, and by

[1]In our limit study, we assume that the execution can use as many service cores as required. Later, we relax this assumption, and investigate a more realistic implementation.

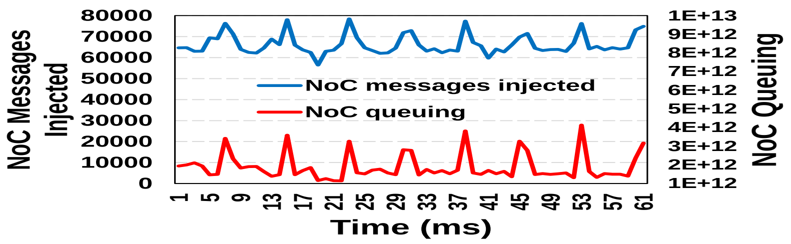


Figure 5.3: Network activity (message injection vs. queuing latency) over time for Histo.

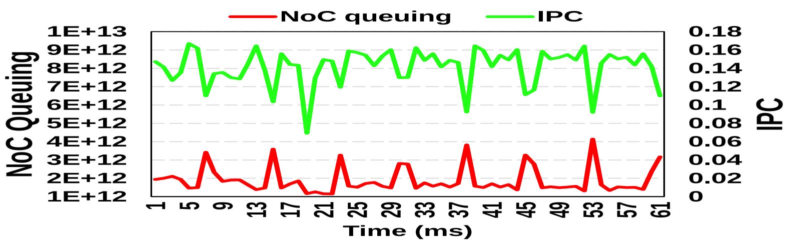


Figure 5.4: Network latency effect on IPC for Histo over time.

contention with other messages in the router queues. Figure 5.2c shows the portion of data access latency contributed by various components of the uncore (NoC and memory). As expected, DRAM access time (steps 2 and 3 in the high-level view shown in Figure 5.2(b)) is the largest portion; however, we observe that NoC latency (which is a measure of data movement) is also a significant contributor to the overall latency (about 26%). For this reason, *our proposed near-data computing approach tries to avoid sending messages in the network as much as possible.* Figure 5.3 plots how the NoC queuing latency varies with the number of messages injected to the on-chip network for histo, one of our applications. As expected, NoC queuing latency increases with the increase in the number of messages injected into the network. Figure 5.4 on the other hand shows how the IPC varies for the same application with the NoC queuing latency over time. As can be observed, IPC comes down with the increase in the NoC queuing and increases as the NoC queuing decreases. Thus, one can conclude from Figures 5.2c, 5.3 and 5.4 that the number of messages and hence NoC queuing plays a significant role in the performance of this application. Similar observations are made in the other target applications as well.

5.4 Evaluated Kernels

In this section, we describe the various kernels that were used in our evaluation. Each of these kernels is used as a “target application” on our simulation.

Algorithm 5 Sum of Absolute Differences

```
 1: for each macroblock do
 2:     for each search position do
 3:         for each 4-by-4 block do
 4:             sad ← 0
 5:             for each pixel do
 6:                 a ← pixel from image A
 7:                 b ← pixel from image B
 8:                 sad ← sad + (a - b)
 9:             end for
10:         end for
11:     end for
12: end for
```

Algorithm 6 7-point Stencil

```
1: for i = 1 to nx-1 do
2:     for j =1 to ny-1 do
3:         for k = 1 to nz-1 do
4:             temp ← A0[i,j,k+1] + A0[i,j,k-1] + A0[i,j+1,k] + A0[i,j-1,k] + A0[i+1,j,k] + A0[i-1,j,k]
5:             temp ← temp × c1
6:             $A_{next}$ ← temp - c0 × A0[i,j,k]
7:         end for
8:     end for
9: end for
```

Algorithm 7 Scan

```
1: reference[0] = data[0]
2: for i = 1 to size do
3:     $reference[i] = data[i] + reference[i-1]$
4: end for
```

5.4.1 SAD

Sum of Absolute Differences (SAD) is a common kernel used in scientific as well as multimedia applications. It is used to find out how well two images are correlated and is also widely used in video encoding in benchmarks such as disparity from

San Diego Vision Benchmark Suite [82] and X264 in Parsec [83]. This kernel, given as Algorithm 5, goes through the entire image pixel-by-pixel and finds the sum of absolute differences of corresponding pixels and hence is very data-intensive. As it involves simple operations such as subtraction (for finding the difference) and multiplication (for squaring), this is a very good candidate for offloading. The version we use is from Parboil [84].

5.4.2 Stencil

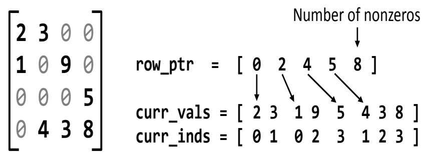
Stencil computations are very common and are often found in computer simulations for fluid dynamics, image processing, cellular automata and also for solving partial differential equations (PDEs) such as the Jacobian method, etc. These computations can operate on 1D, 2D and 3D grids depending on the application. A computation involving n different points (a center point and its neighbors) called a n-point stencil. This kernel has a regular access pattern, but updates to an element access data in multiple dimensions, which can be distant in memory. In this work we evaluate a data-intensive 7-point stencil on a 3D grid, shown in Algorithm 6. This code is also from Parboil [84].

5.4.3 Scan

Scan is a common benchmark in most of the scientific kernels. It performs the all-prefix-sums operation, which calculates the sum of an element and all previous elements at each point in an array. This kernel, given as Algorithm 7, is widely used in applications such as sorting, polynomial evaluation, string comparison, and many others. Because of its regular, sequential access pattern, Scan has good locality. However, because of the dependence between `reference[i]` and `reference[i-1]`, it is not simple to parallelize. We use the implementation from Scalable Heterogeneous Benchmark Suite (SHOC) [85].

Algorithm 8 Sparse Matrix-Vector Multiplication

```
1: for each row in matrix do
2:     sum = 0.0
3:     for each nonzero in row do
4:         sum ← sum + cur_vals[j] * x[cur_inds[j]]
5:     end for
6: end for
```



Number of nonzeros
row_ptr = [0 2 4 5 8]
curr_vals = [2 3 1 9 5 4 3 8]
curr_inds = [0 1 0 2 3 1 2 3]

Figure 5.5: SpMV compressed sparse row matrix format. `row_ptr` contains the list start indices for each row. `curr_vals` contains values of the nonzero elements of the matrix. `curr_inds` is a parallel array that contains the corresponding column index of each nonzero element.

Algorithm 9 Histogram

```
1: for each pixel in image do
2:     value ← img[i]
3:     if histo[value] < 255 then
4:         $histo[value]$++;
5:     end if
6: end for
```

5.4.4 SpMV

Sparse Matrix Vector multiplication (SpMV) is a very commonly used scientific kernel. We use the SpMV kernel from HPCCG in Mantevo [46]. This application uses a compressed sparse row (CSR) format for matrices, shown in Figure 5.5. The kernel given in Algorithm 8, has an irregular access pattern because the elements of one array (`cur_inds`) are used as indices into another array (`x`, the column vector). This access pattern is data-dependent and very difficult to analyze at compile time.

5.4.5 Histogram

The Histogram kernel, shown in Algoirthm 9 calculates the frequency of each color in an image. This kernel is used to aid in image editing and in computer vision, for example, in image segmentation. Like SpMV, it contains a data-dependent irregular access where the value of the `img` array is used as an index into the `histo` array. The version of histogram evaluated in this work is from Parboil [84].

5.5 Experimental Evaluation

In this section, we briefly explain our experimental setup. In all these simulations, the service cores are assumed to be a Simultaneous Multi-threaded (SMT) core, where the co-runner application and the off-loaded computation (from the target application) run in separate SMT ways. In other words, these service cores are specialized cores with SMT such that they can run the co-runner application and the offloaded computation simultaneously.

5.5.1 Benchmarks

Table 7.2 summarizes the applications used in our evaluations. The kernels described in Section 5.4 are our "target applications" , while the "co-runner applications" are used to fill the other cores on the chip. The co-runner applications (astar, mcf, xalancbmk, povray, and omnetpp) are from SPEC CPU2006 suite [44]. Using our target and co-runner applications, we formed different workloads to carry out our study. These workloads are given in Table 6.2. We evaluated 11 workloads each of them contain one target application running along with co-runner applications. These co-runner applications are chosen such that they stress the NoC and memory resources differently. These co-runner applications consist of mcf (very highly memory-intensive), xalancbmk (memory-intensive), omnetpp (moderately memory-intensive), and povray (not memory-intensive).

5.5.2 Simulation Platform

We use Sniper [86] as our evaluation platform to perform the limit study. We simulate a Gainestown X86 Intel processor in System Emulation mode with 32 cores running at 2.66 GHz and an 8x4 mesh-based network on-chip interconnect. In our simulations, we fast-forward the simulation up until the benchmarks are done reading from the input files. We then simulate from the start of the region of interest until the entire target kernel finishes execution. The salient characteristics of our default system simulated in Sniper are listed in Table 7.1.

5.6 Ideal Near-Data Computing Strategies in Manycores

As stated earlier, in this work, we study the potential of three different versions of near-data computing in manycores. In this section, we first describe our ideal near-data computing strategies, and then show experimental results using these

	Application	**Description**	**Input Size**
Target Applications	Histogram	Histogram of image pixel values	2×4MB
	SAD	Sum of Absolute Differences of two images	2×4MB
	Scan	Prefix sum	2×16MB
	Stencil	3D 7-point Stencil	4MB
	SpMV	Sparse Matrix-Vector multiplication	$50 \times 50 \times 50$
Co-runner Appllications	astar	Pathfinding for 2D maps	ref
	mcf	Vehicle scheduling with network simplex algorithm	ref
	xalancbmk	XML processing/conversion	ref
	povray	Image ray-tracing	ref
	omnetpp	Discrete network event simulation	ref

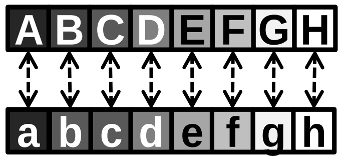
Table 5.1: Applications used in the experiments.

Workload	**Application Mix**
WL 0	target app, mcf (31)
WL 1	target app, xalancbmk (31)
WL 2	target app, povray (31)
WL 3	target app, astar (31)
WL 4	target app, omnetpp (31)
WL 5	target app, mcf (1), xalancbmk (5), povray (9), astar (7), omnetpp (9)
WL 6	target app, mcf (12), xalancbmk (5), povray (2), astar (5), omnetpp (7)
WL 7	target app, mcf (1), xalancbmk (16), povray (2), astar (5), omnetpp (7)
WL 8	target app, mcf (1), povray (5), astar (6), omnetpp (19)
WL 9	target app, mcf (1), povray (18), astar (5), omnetpp (7)
WL 10	target app, mcf (6), xalancbmk (3), povray (5), astar (13), omnetpp (4)

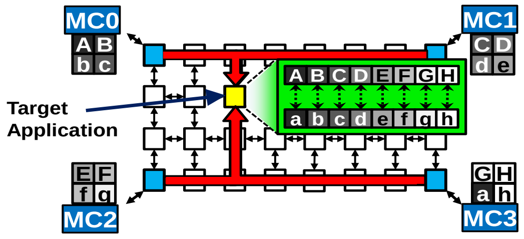
Table 5.2: Workloads used in the experiments. The numbers within parantheses indicate the number of instances of that application in the workload.

schemes. Note that, a perfect near-data computing system should incur 0 latency when the computations are offloaded to the service cores. Also, results from the offloaded computations should not incur any network latency and should not incur additional latency due to the contention for computational resources at the service cores.

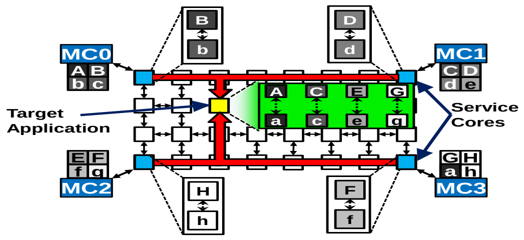
Table 5.4 gives a summary of these versions, which we explain below using SAD as a running example. Suppose we have two images with 8 pixels each (or blocks) to be compared. Figure 5.6(a) shows how each pixel in the top image (capital letters) correlates to a pixel in the bottom image (lowercase letters). The SAD kernel will loop through both arrays of pixels and compute the absolute value of the difference of each pair.



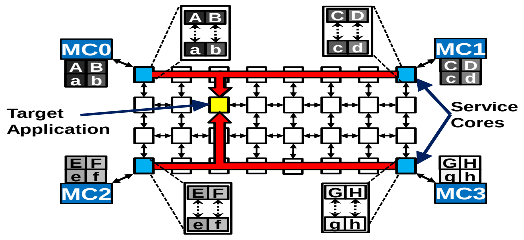
(a)



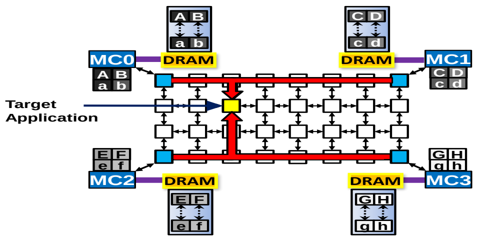
(b)



(c)



(d)



(e)

Figure 5.6: Sum of Absolute Differences (SAD) example (a) Pairs of pixels from two images to be compared (default). (b) baseline execution. (c) NDC-1. (d) NDC-2. (e) NDC-3.

Co-runner Core	22nm Ivy Bridge, 2.66 GHz
Service Core	22nm Ivy Bridge, SMT Support, 2.66 GHz
L1 Cache	32KB, 8-way associative, access latency: 4 cycles
L2 Cache	256KB, 8-way associative, access latency: 8 cycles
L3 Cache	512KB per core (shared), 16-way associative, access latency: 30 cycles
Coherence	MESI
NoC	8x4 (32 cores), hop latency: 2 cycles, 32 bits/cycle
Memory	DDR3-1600 DRAM, latency: 30 ns, 8KB Page, 4 Channels, Ranks per Channel: 4, Banks Per Rank: 8

Table 5.3: Simulated architecture specification

Version	Placement		Savings			Hardware Changes
	Computation	Data	Memory Bandwidth	NoC Queuing	NoC Hop + Processing	
Baseline	application core	default	-	-	-	none
NDC-1	service cores, if possible	default	-	yes (offloaded data)	yes (offloaded data)	none
NDC-2	service cores	perfect	-	yes	yes	none
NDC-3	DRAM (off-chip)	perfect	yes	yes	yes	PIM
NDC-4	service cores	perfect	-	yes (EVC links)	yes (EVC links)	EVCs
NDC-5	service cores	perfect	-	yes (EVC links)	yes (EVC links)	EVCs
NDC-6	service cores	perfect	-	yes (fast NoC links)	yes (fast NoC links)	High Freq. Routers
NDC-7	service cores	perfect	-	yes (fast NoC links)	yes (fast NoC links)	High Freq. Routers

Table 5.4: Summary of the near-data computing strategies evaluated in this work. Note that NDC-1 only saves NoC latency for the fraction of computations that were offloaded, and NDC-4 and NDC-5 save NoC latency only for EVC covered links.

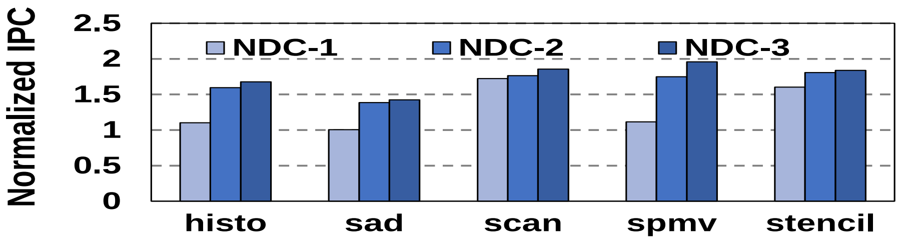


Figure 5.7: Normalized IPC for the perfect schemes.

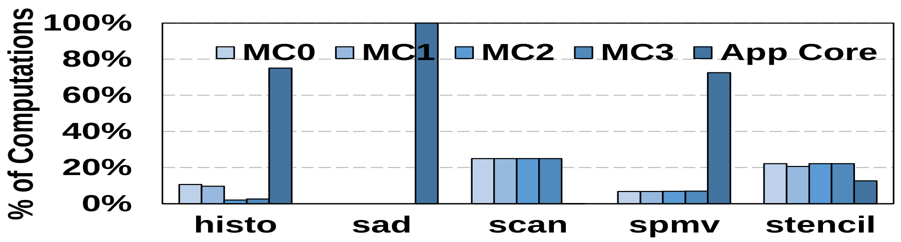


Figure 5.8: Location of computation for each target application under NDC-1.

5.6.1 Description of Near-Data Computing Strategies

Our **baseline** (also called basecase) uses the default data placement (determined by the OS) and runs all computations on the target application core. All data used in the computation must be sent over the network. An example of this conventional execution is illustrated in Figure 5.6(b). For the pair of corresponding data elements A and a, A will come from MC0 (3 hops) and a will come from MC3 (7 hops), after which the application core will compute $|A - a|$ in one iteration of the kernel. In this case, all 16 data elements needed for the complete execution will be sent from the MCs over the network to the target application core in the center of the chip, and all eight absolute differences will be computed there. Finally, all the differences will be added together to produce the final sum.

NDC-1: Our first near-data computing strategy is called NDC-1. This scheme uses the same default data placement (i.e., it does not modify the original data-to-memory controller assignment), but tries to offload some of the computations to the service cores near the memory controllers. Note that, computation will only be offloaded if *all* data needed by that computation is assigned to the same memory channel. If so, the computation will be moved to the core immediately next to the MC for that channel. If the data items reside in multiple channels, they will be sent over the network and used in the target application core as in the original case. In our evaluation, the potential of this version of near-data computing, offloaded computations incur no network queuing, transmission, or processing latency.

NDC-2: In NDC-1, the performance is limited by the default (imperfect) data placement. Our second strategy, called NDC-2, assumes that all the data can be assigned to memory channels *perfectly* such that *every* computation can be run on the service cores. Figure 5.6(d), shows how perfect data placement can greatly reduce the network traffic; in this example, it is possible to do all the computations in the service cores near the memory controllers. Since there are multiple pairs of pixels being subtracted in each service core, the addition will also be partially offloaded. For example, in MC1, the partial sum $|C-c|+|D-d|$ will be computed. Finally, only the four partial sum results need to be sent to the application core, greatly reducing the application's network traffic. In quantifying the potential of this scheme, we assume that all computations are offloaded and incur no network latency.

NDC-3: Finally, we consider the extreme case where all computations are further offloaded to the DRAM, called NDC-3. In this case, the perfect data placement is also used. In addition to saving network latency, in our evaluation, we assume that computations in the DRAM do not incur any memory queuing latency. An example of NDC-3 is shown in Figure 5.6(e). As in NDC-2, none of the original operands need to be sent over the network.

Table 5.4 summarizes all of the strategies evaluated in this work, including the placement of computation and data, which parts of memory access latency are saved, and any extra hardware required. Note that NDC-1, NDC-2, and NDC-3 can save NoC latency, while NDC-3 is the only scheme that can also save memory bandwidth[2]) . NDC-4 through NDC-7 will be discussed in Section 5.7.

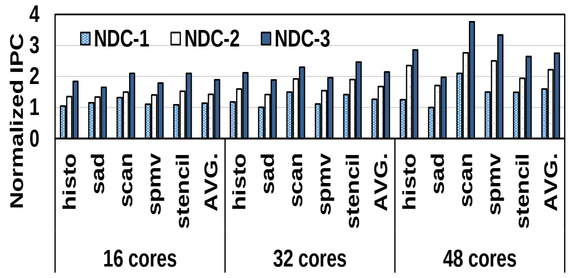
5.6.2 Results with Single-threaded Target Applications

The IPC results normalized with respect to the baseline (original execution without any near-data computing) are summarized in Figure 5.7 [3]. For Histogram, the IPC value for NDC-1 for is very close to the baseline, i.e., there is little improvement. As shown in Figure 5.8, most of the computations in Histogram (about 75%) are performed on the application core (not on the service cores) because they were not able to be offloaded. This is not surprising considering the irregular access pattern caused by indirect array accesses for Histogram.

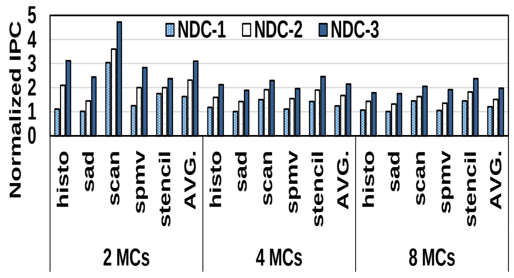
In the case of Scan, NDC-1 brings a large improvement over the baseline and approaches NDC-2 in terms of performance. We can see from Figure 5.8 that most

[2]To determine the best scheme suitable for a workload dynamically, the OS buddy allocator which allocates data can be enhanced to take hints such that the memory channel bits in both the virtual address and the physical address are the same. Hence if the virtual addresses of variables used in a computation are mapped to the same memory controller, NDC-2 can be triggered, else the other schemes can be triggered accordingly.

[3]Unless otherwise stated, all IPC (performance) numbers are normalized with respect to the *baseline* execution.

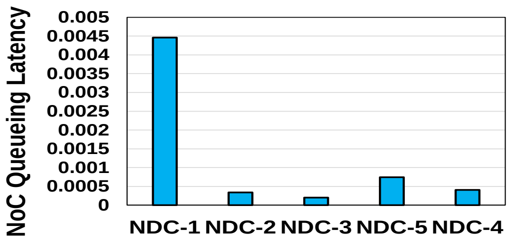


(a)

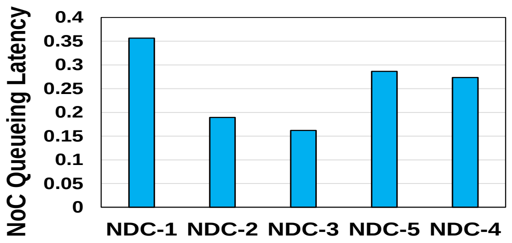


(b)

Figure 5.9: Normalized IPC Improvement with varying (a) core counts. (b) memory channels.



(a) Scan



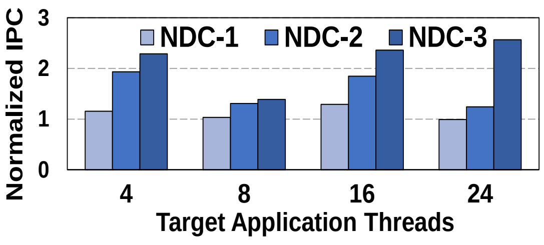
(b) Stencil

Figure 5.10: Normalized NoC queuing latency with respect to baseline for Scan and Stencil applications.

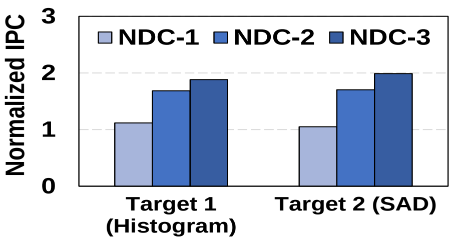
of the computations in Scan are successfully offloaded to the service cores, even under the default data placement. Scan has a very regular access pattern and good locality. The performance improvements for Scan are also supported by the reduction in NoC queuing latency, shown in Figure 5.10a. Compared to baseline, there is almost no queuing latency for the core running Scan.

The results for Stencil in Figure 5.7 show that NDC-1 gives good improvements, but there is still room for improvement from NDC-1 to NDC-2 and NDC-3. Only about 13% of the computations for Stencil could not be offloaded to the service cores. Also, observe in Figure 5.10b that the queuing latency reductions for Stencil are more modest than for Scan. This application has reasonably good locality and a regular access pattern, but some vertical accesses in the 3D grid may come from different memory channels.

In the case of SPMV and Histo, since their codes involve irrregular data accesses, as can be observed from Figure 5.8, a very low percentage of the data falls into the same memory channel and consequently, NDC-1 achieves very low improvements in performance. Note that these applications have very low locality and incur lot of memory accesses. However, NDC-2 and NDC-3 show significant improvements



(a)



(b)

Figure 5.11: (a) Normalized IPC with varying numbers of threads for SpMV, and (b) shows IPC for the multi-target workloads.

in performance for both the benchmarks as their queuing latency goes down with the offloaded computations. For the SAD benchmark, the two images compared for a given data placement are not aligned completely and hence NDC-1 does not bring any performance benefit compared to the baseline. Compared to other target applications, SAD has relatively low memory pressure because of the high spatial locality and, as can be observed from Figure 5.7, the average performance improvements are relatively low compared to other applications.

5.6.3 Multi-threaded Target Application Results

In the multi-threaded target scenario, we have pinned the threads to consecutive cores on the manycore and the co-runner applications run on the other cores. As we increase the number of threads for the target applications, the number of co-runner applications decreases; however, the memory pressure is still maintained as now all the threads of the target application simultaneously request unshared data. SpMV scales well with number of threads in the baseline. We present IPC results with varying number of threads (4, 8, 16 and 24) in Figure 5.11(a). The co-runner application in all these experiments is mcf from SPEC 2006 which is highly memory-intensive. As expected, the in-memory computation (NDC-3) yields the best performance as it has to only send back the results and does not require high memory bandwidth. However, for the multi-threaded case, the gap between the in-memory computation and the perfect data placement case widens because, as thread-level parallelism increases, the memory queuing dominates. Since this queuing latency is not incurred in the case of in-memory computation, the performance is better with 16 and 24 threads. NDC-2 still yields significant improvements.

5.6.4 Results with Multiple Target Applications

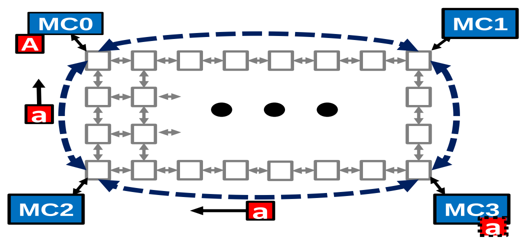
We have also experimented with multi-targeted workloads, with histo and SAD applications pinned to different cores. We present IPC results in Figure 5.11(b). The co-runner application in all these experiments is mcf from SPEC 2006 which is memory-intensive. As expected, the in-memory computation version (NDC-3) yields the best performance as it has only to send back the results and does not require high memory bandwidth. However, for the multi-threaded case, the gap between the in-memory computation case and the perfect data placement case widens because as thread-level parallelism increases, the memory queuing dominates. Since this queuing latency is not incurred for in-memory computation, the performance is better with 8 and 16 threads.

5.6.5 Results with Varying Core Counts

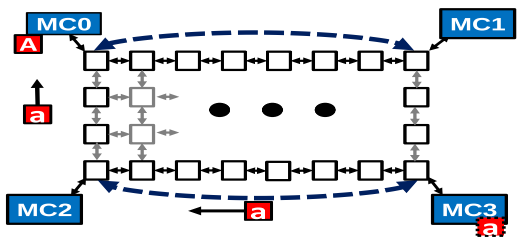
Results with varying core counts are shown in Figure 5.9(a). As can be observed, performance benefits from 32 cores are higher than that of 16 cores while performance benefits with a 48 core configuration are much greater than that of 16 core and 32 core configuration. This shows that offloading computations to service cores scales very well with increasing core counts. This is primarily because as the core count increases, the congestion in the network-on-chip increases resulting in increased NoC queuing. Since offloaded computation minimizes the number of packets traversing the network, performance benefits are significant. We expect the trend to continue with much higher core count.

5.6.6 Results with Varying Number of Memory Channels

Results with varying number of memory channels are plotted in Figure 5.9(b). As can be observed, the performance improvement diminishes with the increased number of memory channels. This is because, with the increased number of channels, the number of hops to be traversed by each request before getting to the off-chip memory decreases. As a result of this decrease, the NoC latencies for each request decrease (especially the queuing latency). Since our schemes aim at reducing the NoC latencies, increasing the number of channels reduces the potential improvements considerably.



(a)



(b)

Figure 5.12: Express virtual channels. (a) System with 4 EVCs shown in blue, no NoC queuing latency is incurred and used for NDC-4 (b) System with 2EVCs (in blue), NoC queuing latency incurred for vertical hops and used for NDC-5.

5.7 Realistic Near-Data Computing Strategies

In the previous section, we show results with a perfect (near-data computing) system with infinite hardware resources at the service cores and zero on-chip latency for offloaded computations. In this section, we relax these assumptions and evaluate the resulting performance benefits. Our goal is to measure how closely one can approach the results in the previous section by employing different architectural enhancements. We discuss various extra hardware resources which are needed to approximate the perfect near-data computing systems discussed so far.

5.7.1 Employing Express Virtual Channels

Express Virtual Channels (EVCs), proposed in [81], incur minimal additional latency of 2 cycles compared to an ideal latency system. With EVCs in place, the on-chip requests do not incur any queuing latency in the network on-chip buffers and only incur the packet transmission and router processing latencies. Furthermore, EVCs were proven to be deadlock-free and yield good improvements in performance. However, EVCs require an extra overheads in terms of area and power. Since they are the first step towards realizing a near-perfect system for on-chip near-data computing, we experiment with varying number of EVCs and report the performance results.

Figure 5.12(a) shows the configuration with four EVCs connecting the four corners of the chip directly. Suppose that A and a are needed for the same computation, but are in different memory channels. a can easily travel to A's service core using the EVCs. Only the transmission latency is considered for messages using these EVCs. Using this system, we create a new version, NDC-4,

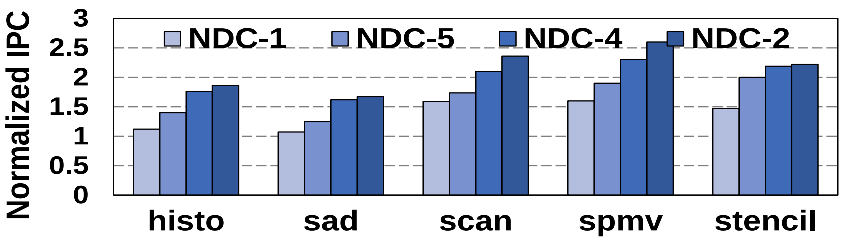


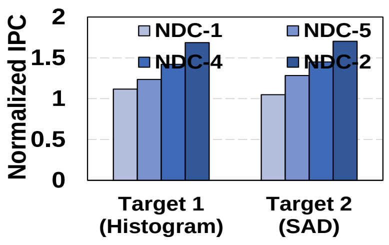
Figure 5.13: Normalized IPC for the relaxed schemes. NDC-1 and NDC-2 are shown for comparison.

which, similar to NDC-2, uses perfect data placement and computation on service cores, but with the *added NoC transmission latency*. Figure 5.12(b) shows a modified version in which messages can use express channels for the horizontal hops, incurring only transmission latency, but use the normal on-chip routers for the vertical hops, where both queueing latency and transmission latency are added. This scheme is called NDC-5. Like NDC-4, it is a more realistic version of NDC-2 with perfect data placement and computation offloading.

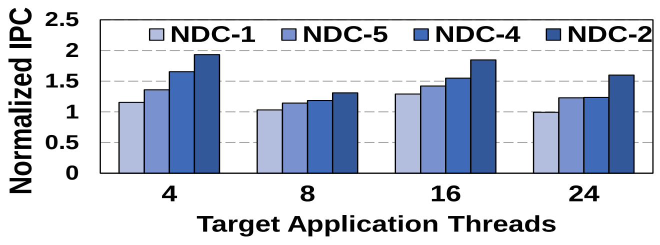
Figure 5.13 shows the IPC results for each target application under these two relaxed schemes, NDC-4 and NDC-5. NDC-1 and NDC-2 are shown for comparison because NDC-2 represents in a sense an idealized version of these two schemes, and NDC-1 represents the performance with some computation offloading but *less-than-perfect* data placement. As can be observed, using 4EVCs (NDC-4) brings better performance improvements compared to using 2EVCs (NDC-5) for all the target applications. This is because NDC-5 incurs extra NoC queuing latency for the vertical hops compared to NDC-4. This can also be verified from Figure 5.10a and Figure 5.10b, for applications scan and stencil, respectively. We use the same multithreaded target application as in the perfect system, SpMV, to perform experiments with multiple threads. Using the same multi-target workload as in the perfect system, we also present results for the relaxed schemes. Figure 5.14 shows the IPC values normalized to the baseline for two applications on separate cores. For multi-threaded and multi-target workloads, NDC-4 performs better than NDC-5 as latter incurs extra queuing latency for vertical hops.

5.7.2 Employing High Bandwidth Links

In the perfect scenario, explained in Section 5.6, we assumed a perfect NoC which incurs zero latency by employing infinite bandwidth. In this subsection,



(a)



(b)

Figure 5.14: (a) Normalized IPC for multi-target workloads, and (b) IPC with varying numbers of threads for realistic schemes.

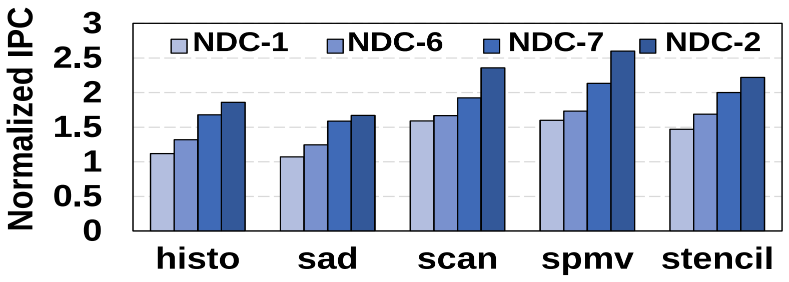


Figure 5.15: Normalized IPC with variation in bandwidth. NDC-1 and NDC-2 are shown for comparison.

we relax that constraint by assuming a finite bandwidth in the NoC links and this bandwidth is modulated by controlling the NoC frequency. Specifically, we change the bandwidth of each link by changing the frequency of the NoC router. Figure 5.15 plots the improvement in performance as we increase the bandwidth of each link. The NDC-6 version whose results are shown in Figure 5.15 provides 48 bits/cycle NoC bandwidth as opposed to the baseline of 32bits/cycle, while NDC-7 provides 64bits/cycle NoC bandwidth. As can be observed, while this technique gives significant improvements compared to the baseline, it does not completely eliminate the NoC queuing, router processing and hop latencies, as in the case of Express Virtual Channels, and hence does not yield as much benefit as EVCs. On the other hand, the advantage of such a technique is that we do not have to add extra hardware like in the case of EVCs to get near-perfect improvements. Adding extra virtual channels not only brings in an area overhead, but also brings in extra overheads of ensuring that the added virtual channels do not result in deadlock/livelock scenarios. Also, increasing the router frequency has its own advantages, as it comes with power/thermal overheads, which need to be carefully analyzed.

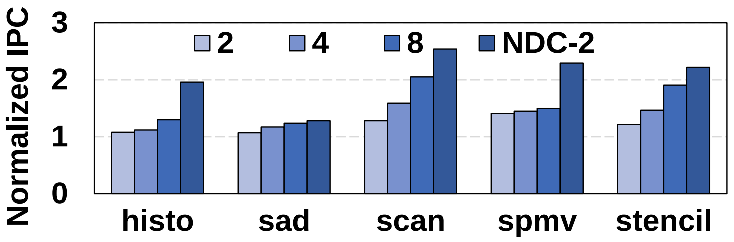


Figure 5.16: Normalized IPC with variation in the number of service cores. NDC-2 (perfect scenario) is shown for comparison.

5.7.3 Impact of the Service Core Count

In both the realistic scenarios evaluated above, we have relaxed the assumption of having infinite NoC resources. However, the number of service cores employed to perform the offloaded computations can also play a significant role in getting near-ideal performance. In this section, we quantify the effect of changing the number of service cores on the performance. As can be observed from Figure 5.16, for benchmarks such as sad, scan, and stencil, with increased number of service cores, offloaded computations incur minimum contention at the service cores, and consequently the IPC improvements increase with the increase in the number of service cores. However, other benchmarks such as histo and spmv do not gain as much improvement as the amount of offloaded computations is very low. The extra overhead with this approach is that each service core supports Simultaneous Multi-Threading (SMT), which means that as we increase the number of service cores, the number of SMT ways increases, resulting in an increase in the overall core area. Please note that the computation offloaded to service-cores through a fork-join approach is not a performance bottleneck, since the offloaded instructions executed on the service cores are processor instructions and no-software related overheads are incurred. As can be observed from Figures 5.13, 5.15 and 5.16, number of service cores plays a minimal role in achieving the ideal performance for single-threaded target applications, as compared to the NoC resources.

5.8 Related Work

The concept of moving computation close to the data has been studied since the 1970s. For example, Stone [87] described a design for a logic-in-memory array that would act as a cache and also directly operate on memory locations.

Drapper et al. [88] fabiracted the first smart memory devices that are enabled with Processing in-memory (PIM). Their results showed significant promise for data-intensive applications. In the related area of non-volatile memory, De et al. [89] move computation into the SSD controller to improve the performance of I/O-intensive big data applications.

Pugsley et al. in [77] proposed a NDC architecture to accelerate Map-Reduce workloads by integrating a Near Data Cores (NDCores) on the logic layer of the 3D-Stacked Memory. The NDCores are in-order low-EPI energy-efficient ARM Cortex cores which are optimized for massively parallel execution code-bases like Map-Reduce. While the same authors in [90] augmented the NDC-architecture proposed in [77] by replacing the expensive HMC-based stacked DRAM with LPDDRx powered NDC-Modules. Their evaluation is based on a highly optimized ARM cortex based many-core processor as the baseline. Gao et al. [91] proposed 3D DRAM based Tetris to accelerate machine-learning kernels, while Chi et al [92] employed non-volatile memory based 3D memory to accelerate machine learning kernels. Each chip consists of SDRAM and logic for general-purpose computation and communication. Several PIM chip can communicate independently of the host. While these past designs incur significant improvements in performance, there are major challenges in the overall system design for these architectures. These challenges include: (a) Imposing non-cacheable on-chip cache accesses for the host-processor so that the processing engines on the logic layer of the stacked memories see the coherent data. (b) Enhanced virtual-memory support so that the processing cores can access the data from correct physical locations in DRAM. (c) Pinned physical locations in 3D stacked memories so that the data used by these processing engines on stacked memories are not paged-out by the OS executing on the host-processor. (d) Dynamically identifying the candidate code to be offloaded to the specialised processing engines to reap maximum performance improvement. While Hsieh et al. [76] identified the possible candidate code for offloading dynamically, the other challenges still remain to be addressed, making the prior proposals tough to implement.

Our approach instead of offloading the computations to DRAM, offload the computations to service cores closer to the memory controllers. As a result, our approach targets the on-chip latency instead of the memory bandwidth problem unlike the prior approaches. As a result, our approach allows data accessed by the host-core to cache the data in on-chip caches. As the data accessed by the service core will be taken care by the on-chip cache coherency. Similarly, the on-chip coherent TLBs does not necessitate enhanced virtual-memory support nor the static pinning of memory pages in main memory. By addressing these overall-system design challenges imposed by the prior near-data computing techniques, our proposed hardware-techniques can be readily adapted by industry to mitigate some of the on-chip data movement costs.

To our knowledge, there is no work that tries to move computation into the on-chip network structures such as routers.In comparison to these prior works, ours is the first that quantifies the potential benefits of near-data computing in emerging manycores.

Chapter 6

Co-design to Optimize DRAM Refresh Overheads

6.1 Introduction

Dynamic Random Access Memory (DRAM) is the predominant main memory technology used in computing systems today. DRAM cells use capacitors as data storage devices. Since capacitors leak charge over time, DRAM cells need to be periodically refreshed inorder to preserve data integrity. These periodic refresh operations block main memory accesses, therefore reducing main memory availability and increasing effective memory latency. This problem is even more accentuated in consolidated environments like virtualized systems.

Technology scaling trends have led to continuous increases in DRAM device densities over the last several decades. These scaling trends have enabled higher main memory capacities in all computing segments, paving the path for higher system performance and increasingly sophisticated software. However, as the total number of DRAM cells in a system continues to increase, the DRAM refresh overheads are on the rise and are threatening to dampen the performance benefits of DRAM capacity scaling. Recent studies have shown that for upcoming 32Gb DRAM devices, DRAM refreshes can cause a 30% reduction in overall system throughput [14].

Many recent works have proposed hardware [15] [93] [17] and software [94] [95] solutions to mitigate the performance overheads caused by DRAM refreshes. These approaches can be broadly classified into two categories: (i) reducing the number of refreshes, and (ii) overlapping memory accesses with refreshes. Techniques belonging to the first category reduce refresh activity by refreshing each DRAM row at a different rate, dictated by the cell with the lowest retention time in that row. While these techniques can reduce the number of refresh operations substantially, they rely on accurate retention time profiling, which is costly to implement and is highly prone to erratic changes in DRAM cell retention times [93]. The second category of techniques reduces the exposed refresh overhead by allowing regular DRAM accesses to proceed in parallel with refresh operations. The key idea behind these techniques is to confine the refresh activity to a portion of the DRAM (such as a bank or a subarray), so that refresh operations in one portion will not interfere with accesses to the other (non-refreshed) portions.

The most recent example of such finer-granularity refreshing adopted by the DRAM industry is the per-bank refresh scheme supported in LPDDR3 [52] and beyond. As opposed to the traditional all-bank refresh scheme in earlier LPDDRx generations (and current DDRx generations), a refresh command in the per-bank refresh scheme targets only one DRAM bank. Therefore, while a per-bank refresh command is busy refreshing rows in one bank, all the other banks are available to service regular DRAM accesses. In an ideal scenario, if all the DRAM requests that arrive at the DRAM controller during a refresh operation are headed to the available (non-refreshed) banks, then the refresh overhead can be fully hidden. However, in realistic scenarios, since memory requests generated by typical programs are often uniformly distributed across DRAM banks, the probability of a DRAM request being blocked by a per-bank refresh is quite high. Therefore, as shown in prior studies, per-bank refresh is only marginally effective in avoiding the DRAM refresh overheads [15].

In this work, we propose a hardware-software co-design technique to mitigate the DRAM refresh overheads. Our technique exposes per-bank refresh to the operating system (OS) with the goal to enable higher overlap between refresh operations and regular memory accesses. The key idea behind our technique is to incorporate DRAM bank awareness and refresh schedule in the memory allocation and task scheduling decisions made by the OS. Specifically, our technique proposes the following two main changes to the operating system: (i) the OS memory allocator confines the memory allocated by a task to a subset of the available DRAM banks, and (ii) the OS task scheduler chooses the tasks scheduled during a quantum in such a way that the memory accesses made by these tasks do not span all the DRAM banks in the system. Furthermore, our technique proposes the following change to the refresh scheduler in the memory controller: rather than doing a round-robin scheduling of refresh commands to individual banks, the memory controller refreshes only those banks during a task scheduling quantum which are not expected to receive any memory requests during that quantum. With this careful collaboration between OS and the memory controller, our technique reduces the probability of per-bank refreshes interfering with regular DRAM accesses.

Extensive evaluations of our proposed technique on multi-programmed SPEC CPU2006 [44], STREAM [96] and NAS [97] workloads show that our technique achieves 16.2% and 6.3% performance improvement over all-bank and per-bank refresh for 32Gb DRAM chips, respectively. Our results also show that the co-design improves the performance by 14.6% and 6.1% on an average compared to previously proposed Adaptive Refresh (AR) [16] and per-bank Out-Of-Order refresh [15] respectively, without necessitating any modifications to the internal DRAM structures.

6.2 Motivation

6.2.1 Performance Degradation due to DRAM Refresh

As explained in Section 2.1.3.2, since all banks in a rank are not available during refresh, all-bank refresh is more detrimental to performance compared to per-bank refresh since in the latter only one bank will be refreshed during a refresh interval. Figure 6.1 shows the performance degradation for different DRAM device densitities. As can be observed, for operating temperatures below 85 deg C where the DRAM retention time (tREFW) is 64 msecs, as the chip density increases from 8Gb to 32Gb, performance degrades from 5.4% to 17.2% for all-bank refresh on an average. However, for per-bank refresh, the degradation on an average varies from 0.24% to 9.8%. This shows that refresh becomes much of a problem with growing DRAM densities since tRFC, the refresh cycle time increases from 350nsec for 8Gb to 890nsec for 32Gb device densities. Also, as device density increases from 8Gb to 32Gb, per-bank refresh also degrades performance significantly, by as much as 9.8%

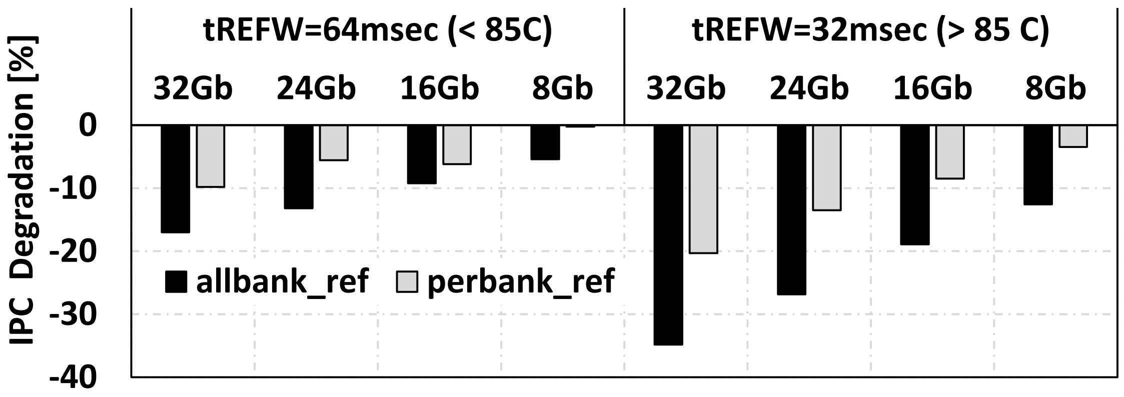


Figure 6.1: Performance degradation due to refresh.

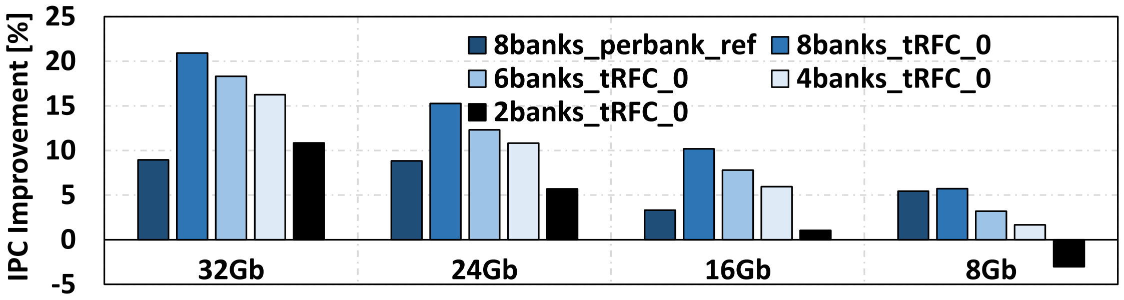


Figure 6.2: IPC Improvements for various DRAM densities, normalized to a scenario where each application uses all the 8 banks in a rank.

as can be observed from Figure 6.1.

DRAM refresh is much more detrimental to performance when the operating temperature is beyond 85 deg C, where the retention is 32 msecs, meaning the DRAM rows need to be refreshed twice as frequently. As can be observed from Figure 6.1, all-bank refresh degrades the performance by up to 34.8% for 32Gb chips on an average, while per-bank refresh degrades performance by up to 20.3%. This shows that DRAM refresh is an important problem that needs to be addressed for the future DRAMs with growing chip densities. The performance degradation due to refresh is expected to be even more pronounced in multi-programmed workloads where multiple high memory-intensive applications are often executed concurrently.

6.2.2 Refresh Cycle Time (tRFC) vs Bank Level Parallelism (BLP)

As explained in Section 2.2.1, the traditional Linux OS is agnostic of DRAM bank organization and allocates data for applications which span across all the DRAM

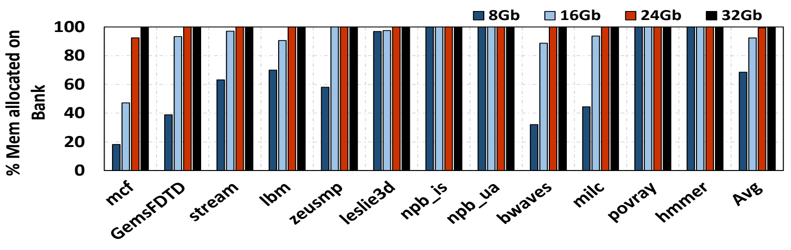


Figure 6.3: Percentage of memory that can be allocated on a single bank with increasing DRAM chip densities (normalized to total memory footprint).

banks. A positive side-effect of such an allocation scheme is increased bank level parallelism (BLP). In our scheme, since the OS partitions memory across tasks, it is important to understand how partitioning an application to access a subset of banks effects performance. Memory-partitioning by OS can increase the DRAM row-buffer locality for certain applications as there will not be any interference from other applications. Since our hardware-software co-design requires partitioning applications' data across DRAM banks and our scheme will ultimately remove the entire tRFC overheads, we present results of various such scenarios in Figure 6.2. As can be observed from this figure, confining applications to a subset of available banks still yields better performance compared to the all-bank refresh if the entire tRFC overheads can be eliminated. Furthermore, confining applications to a maximum of 4 banks per rank (total of 8 banks per channel) still yields improvement in performance in future (16Gb, 24Gb, 32Gb) high-density DRAM chips. However, currently-available density of 8Gb with a lower tRFC, confining an application to few banks degrades the performance as expected, since the BLP is reduced.[1] This result shows that confining applications to a subset of DRAM banks can still yield significant improvements in performance if the entire DRAM refresh related overheads are eliminated.

6.2.3 Feasibility of Bank-Partitioning from a Capacity Stand-point

Having looked at the performance impact of memory-partitioning, we now evaluate the feasibility of memory-partitioning from a capacity stand-point. Since confining an application to a subset of banks limits the overall memory capacity available

[1]Since per-bank refresh yields maximum benefit in 8Gb chips, we do not consider 8Gb in our experiments in the sub-sequent sections.

for an application, it is important to understand the capacity demands imposed by applications. If an application has high memory footprint, confining its data to a subset of banks will increase the number of page-faults in the system even though there is free memory available in the other DRAM banks. Such page-fault scenarios can cause significant degradation in performance. In this subsection, we evaluate the memory footprints of the SPEC CPU 2006 workloads using reference (large) input datasets and the feasibility of bank-partitioning for these applications from the capacity stand-point. Figure 6.3 shows the percentage of memory that can be allocated on each bank with different chip densities, normalized to the total footprint of each application. These results are collected by modifying the default Linux kernel buddy memory allocator, such that the kernel tries to allocate the maximum amount of memory on bank-0. If this cannot be done after a while, the fall-back mechanism would allocate data on other banks using the buddy memory allocator.

Figure 6.3 indicates that for the current-day DRAM chip density of 8Gb, on an average, 68% of applications' total footprint can fit into a single bank. And, this percentage of footprint that can be fit in a single bank increases with the increase in chip density, as can also be noted from Figure 6.3, making bank partitioning-based memory allocator more and more feasible from a capacity stand-point.

6.3 Overview of Our Problem

6.3.1 Problem

Figure 6.4a depicts the modern day dual-core system, two cores C-0 and C-1 executing four tasks T0 - T3 (each denoted by a different pattern). As explained in Section 2.2.1, Linux allocates data for these tasks in a DRAM-oblivious fashion and hence the data for each task are allocated across all the DRAM banks. In Figure 6.4a, the memory allocated for each task by the OS is depicted with the same pattern as the task itself. Consequently, all the tasks T0-T3 can access data from any of the DRAM banks B0 - B3. Figure 6.4b shows the implications of all-bank refresh on a conventional system. Since none of the banks in a rank are available to serve the on-demand requests in all-bank refresh, the probability of the tasks T-0 and T-2 waiting on the data from the banks B0 - B3 is high. Figure 6.4b

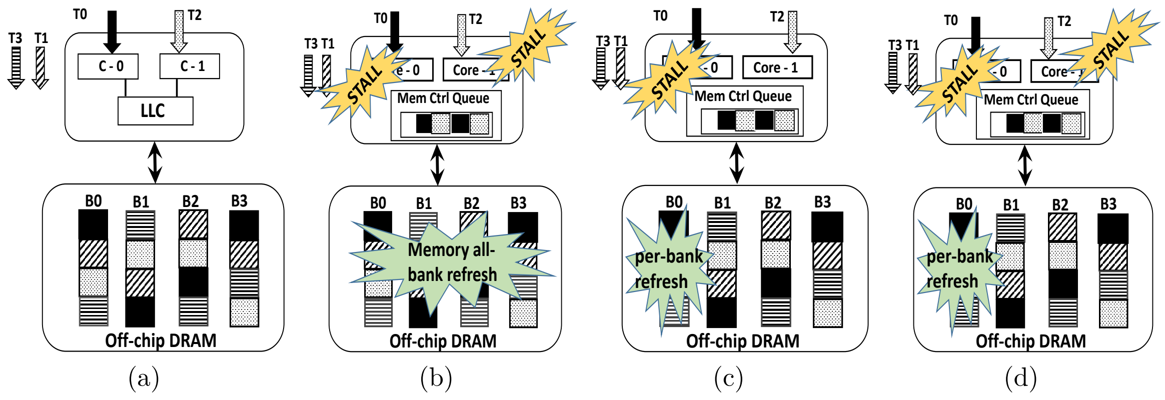


Figure 6.4: Implications of different refresh mechanisms on applications. Diagrams depict (a) Conventional dual-core system executing 4 tasks, (b) worse-case scenario where cores stalled due to all-bank refresh, (c) Not-so-worse scenario where only one core is stalled due to per-bank refresh, and (d) worse-case scenario where both cores can get stalled due to per-bank refresh.

depicts such a scenario where cores C-0 and C-1 are stalled on the outstanding loads (depicted in the MC queue) to be served by the banks being refreshed. However, for per-bank refresh, since only one bank will be busy refreshing, the probability that both cores stalling due to a bank is low. As depicted in Figure 6.4c, there could be not-so-worse scenarios where only one core could be stalled due to per-bank refresh. However, since data of all the tasks are spread across all the DRAM banks, the worst-case scenario of both cores stalling due to a refreshed bank is still possible as depicted in Figure 6.4d. Hence, as observed in Section 6.2.1, allbank-refresh is more detrimental to performance compared to perbank-refresh.

6.3.2 Our Solution

Building on the per-bank refresh support[2], we propose a hardware-software co-design to mitigate DRAM refresh overheads by making changes in both the hardware and the OS. Our proposals are based on the observation that the DRAM retention time (tREFW) and the OS time quanta are in the same order of milliseconds, and include both hardware and software modifications with the goal of eliminating entire DRAM refresh overheads. To this end, we propose a novel and simple per-bank refresh schedule in the hardware which facilitates interesting solutions at the software-level. At the software-level, we use a simple soft-partitioning based

[2]Note that, per-bank refresh already performs significantly better compared to the other prior proposals which are built upon the all-bank refresh strategy as demonstrated in [15].

memory allocator in the OS which augments the proposed per-bank refresh schedule in the hardware. Together, the proposed memory allocator and the proposed per-bank refresh scheduler enable the OS scheduler to schedule the applications in a *refresh-aware fashion*. That is, our proposed hardware per-bank scheduler and soft-partitioning based memory allocator present an opportunity for the OS to schedule an application which does not access the bank being refreshed in it's entire time quantum. This in turn increases the probability that an applications' on-demand requests are not stalled due to refresh.

6.4 Hardware-Software Co-design

6.4.1 Proposed Hardware Changes

Our proposed changes to the per-bank refresh schedule are depicted in Figure 6.5. Comparing Figures 2.4b and 6.5, it can be observed that, in our proposed schedule, in $tREFI_{pb}$-1, instead of refreshing rows R-1 to R-N of Bank-1, we refresh rows R-(N+1) to R-2N. That is, contrary to the default round-robin per-bank refresh scheduler, our per-bank refresh scheduler schedules refreshes to the same bank (to different rows) in successive refresh intervals until all the rows in a bank are refreshed. The pseudo-code for our new per-bank refresh scheduler is given in Algorithm 10.

Algorithm 10 Proposed per-bank refresh schedule algorithm.

```
1: /* nextRefreshBank and the nextRefreshRank represents the bank and the corresponding rank that will be
   refreshed in the next $tREFI_{pb}$. */
2: refreshBankIdx = (nextRefreshRank * numBanksPerRank) + nextRefreshBank
3: /* numRowsRefreshed keeps track number of rows refreshed in a bank.*/
4: numRowsRefreshed[refreshBankIdx] += RowsPerRefresh;
5: if numRowsRefreshed[refreshBankIdx] < numRowsPerBank then
6:     nextRefreshBank = nextRefreshBank;
7: else
8: /* Done refreshing the entire bank. schedule the refresh to the next bank */
9:     numRowsRefreshed[refreshBankIdx] = 0;
10:    nextRefreshBank += 1
11: end if
12: if nextRefreshBank >= numBanksPerRank then
13:    nextRefreshBank = 0;
14:    nextRefreshRank = (nextRefreshRank + 1) % numRanks;
15: end if
```

Implications of our per-bank refresh schedule: Consider a typical system operating in environments below 85 deg C with a tREFW of 64 msec, containing 2

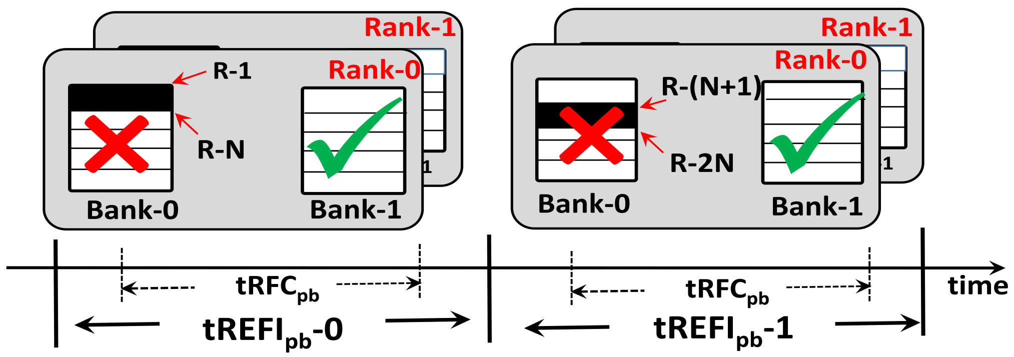


Figure 6.5: Proposed per-bank refresh schedule.

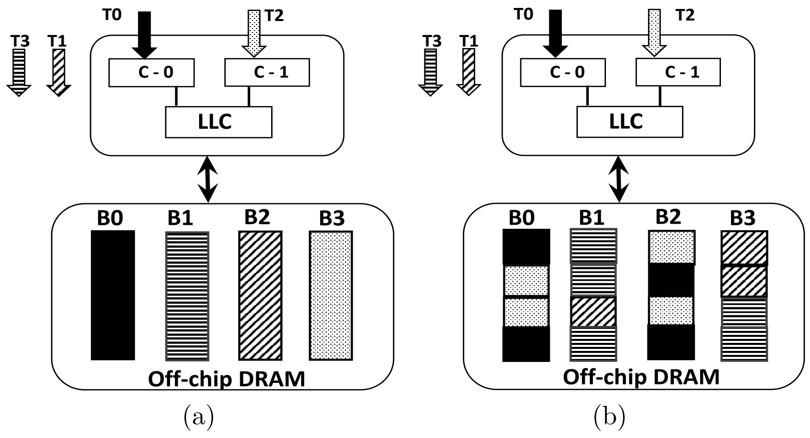


Figure 6.6: (a) Hard-partitioning, and (b) Soft-partitioning based memory allocators.

ranks and 8 banks per rank. In this system, with a total of 16 banks, using our proposed per-bank refresh schedule, all the rows in Bank-0 are done refreshing at the end of first 4msec. Since Bank-0 will be refreshed again only after the 64msec, Bank-0 will be available to serve the on-demand memory requests uninterruptingly after the first 4msecs in a 64msec refresh window. As covered in Section 2.2.2, this duration of 4msec coincides with the process scheduling time quantum used by the OS. Consequently, the new per-bank refresh scheduler enables interesting options for task scheduling in the OS if the applications' memory could be carefully partitioned such that *not* all the banks contain data from all the applications.

6.4.2 Proposed Software (OS) Changes

6.4.2.1 Memory Partitioning Based Allocator:

Various DRAM bank-aware memory partitioning algorithms have been proposed in [20] [19] to alleviate the interference across applications running on different cores. We envision two different ways of partitioning memory as depicted in Figures 6.6a and 6.6b.

Hard-partitioning based allocator: Figure 6.6a shows the hard-partitioning based memory allocator. In such an allocator, each memory bank or a group of banks can host data only from a certain task. As depicted in Figure 6.6a, task T0's data is allocated in bank B-0, task T1's data is allocated in bank B-2, task T2's data is allocated in bank B-3, and T3's data is allocated in bank B-1.[3] Liu et al [19] proposed such a hard-partitioning based memory allocator. By allocating a task's data exclusively on a subset of banks, such an allocator alleviates the memory bank contention, thereby increasing row-buffer locality. However, there are certain drawbacks to such an allocator:

- Confining applications to certain set of banks results in poor bank-level parallelism (BLP) [21] for applications that do not have high row-buffer locality, e.g., irregular applications and pointer-based applications.
- Hard-partitioning can cause a task to page-fault when it is under-provisioned in terms of the number of banks, even if the other banks contain free memory. Such a scenario can be catastrophic to performance.
- With the increasing number of cores on-chip, hard-partitioning limits the overall memory bandwidth available for a task causing the performance to degrade compared to the baseline DRAM bank-agnostic memory allocation.

Soft-partitioning based allocator: An alternative to hard-partitioning is "soft-partitioning" where a group of tasks share a subset of DRAM banks, as depicted in Figure 6.6b. In the soft-partitioning scheme, instead of dedicating a DRAM bank to an application, a DRAM bank is loosely partitioned such that a group of tasks can share it. In Figure 6.6b, tasks T-0 and T-2 have data allocated in banks B-0 and B-2, while tasks T-1 and T-3 have data allocated in banks B-1 and B-3. Hence, such a soft-partitioning based allocator increases the overall memory utilization by

[3]Note that though in this example each task's data is allocated in only one bank, the OS can allocate multiple banks to a task. However, other tasks cannot have data allocated in these banks.

sharing the capacity with other tasks and is more likely to reduce the number of the page-faults in a system. It also caters to the increased BLP, thereby increasing the overall memory bandwidth available for a task at the cost of row-buffer locality.

Algorithm 11 Proposed memory-partitioning algorithm.

```
 1: procedure GET_PAGE_FROM_FREELIST(..., unsigned int order, ..., struct zone *preferred_zone, int migrate-
    type)
 2: /* current –> pointer to the current task which requested the memory allocation */
 3: /* free_list –> original free list maintained by the OS */
 4: /* free_list_per_bank –> per bank free-list */
 5: /* possible_banks_vector –> Bit mask representing bank bits.*/
 6: /* lastAllocedBank represents the bank amongst the possible banks where the last memory request is allocated
    for the current task. */
 7:     for each order in MAX_ORDER do
 8:         count = 0
 9:         for count < num_total_banks do
10:             allocBank = current–>lastAllocedBank;
11:             allocBank = (allocBank+1) % num_total_banks;
12:             if current–>possible_banks_vector[allocBank] is set then
13:                 if free_list_per_bank[bank] is not empty then
14:                     /* Hit from a per bank free list */
15:                     page = free_list_per_bank[bank];
16:                     current–>lastAllocedBank = allocBank;
17:                     return page;
18:                 else
19:                     /* Fetch a page from OS free-list */
20:                     page = list_entry(free_list, ....);
21:                     nr_free--; /* Decrementing the number of OS free pages */
22:
23:                     /* Since OS is exposed with hardware address-mapping information, we can get the bank
    id from the physical page address */
24:
25:                     bank = get_bank_id_from_page(page);
26:
27:                     if allocBank == bank then
28:                         /* Matches the round-robin bank */
29:                         current–>lastAllocedBank = allocBank;
30:                         return page;
31:                     else
32:                         /* Maintaining a cache of per bank free-lists*/
33:                         insert_in_to_free_list(free_list_per_bank, bank, page)
34:                     end if
35:                 end if
36:             end if
37:             count++;
38:         end for
39:
40:     end for
41:     return NULL;
42: end procedure
```

Algorithm 11 shows the detailed pseudo-code for our general memory-partitioning allocator which can either hard-partition or soft-partition data across DRAM banks. We implemented and verified this algorithm in the actual Linux buddy allocator for our experiments. As can be observed from lines 15 and 33, we maintain a free-list of pages per bank so that a free page corresponding to a bank is known readily without traversing the OS free-list. Also, the possible_banks_vector used in line 12 is a bit-mask which represents the possible list of banks that contain data from this particular task. In our current implementation, this possible_banks_vectors bit-mask is an input taken from the user using debugfs [98] and cgroups [99] features in Linux. Hence, our partitioning based allocation presented in Algorithm 11 is generic for both the hard-and soft-partitioning schemes, and can be configured dynamically based on the possible_banks_vector bit-mask. One more important aspect to be noted in our implementation from lines 10-11 is that our memory-partitioning allocator allocates pages such that the consecutive allocation requests[4] fall into different banks in a round-robin fashion by keeping track of lastAllocedBank per task, thereby improving BLP. In our experiments, we observed that soft-partitioning yields better performance as the number of applications running concurrently increases. This is because the memory bandwidth per task increases with soft-partitioning.

6.4.2.2 DRAM Refresh-Aware Process Scheduling:

The new per-bank refresh schedule and memory-partitioning proposed in the previous subsections provide an opportunity for the OS to schedule tasks in a *refresh-aware fashion*. The pseudo-code for the proposed refresh-aware process scheduler is presented in Algorithm 12. As can be noticed in Algorithm 12, nextRefreshBank represents the next bank to be refreshed based on our new per-bank refresh schedule. The code-snippet in the algorithm is the actual implementation in the Linux CFS scheduler, which returns the next task to be scheduled on a core. Our refresh-aware implementation is depicted in line 27 where the next runnable task chosen is the one that does not have any data allocated on the bank which will be refreshed in the next time quantum. This task to be scheduled is one among the tasks to the left in the red-black tree maintained by the CFS scheduler. Hence, by chosing the task which is left among the runnable tasks in the red-black tree, our scheduler tries to schedule a task which does not have any data allocated in the bank to be refreshed.

[4]We do not consider large-pages in our evaluation, hence each allocation granularity is 4KB.

Algorithm 12 Proposed refresh-aware process scheduling.

```
procedure PICK_NEXT_TASK(struct rq *rq)

    /* nextRefreshBank –> represents the next bank to be refreshed in DRAM based on the new per-bank
    refresh schedule */

    struct task_struct *p;
    struct cfs_rq *cfs_rq = &rq–>cfs;
    struct sched_entity *se;
    struct sched_entity *firstSchedEntity;

    if !cfs_rq–>nr_running then
        return NULL;
    end if
    found_task_flag = false;
    count = 0;

    do
        count++;
        se = pick_next_entity(cfs_rq);
        set_next_entity(cfs_rq, se);
        cfs_rq = group_cfs_rq(se);
        p = task_of(se);

        if count == 1 && cfs_rq then
            firstSchedEntity = se;
        end if

        if cfs_rq && p–>possible_banks_vector[nextRefreshBank] is not set then
            found_task_flag = true;
        else if cfs_rq && count >= $\eta_{thresh}$ then
            found_task_flag = true;
            p = task_of(firstSchedEntity);
        end if
    while !flag_found_task;
    / ******* Some more Code ***/
    return p;
end procedure
```

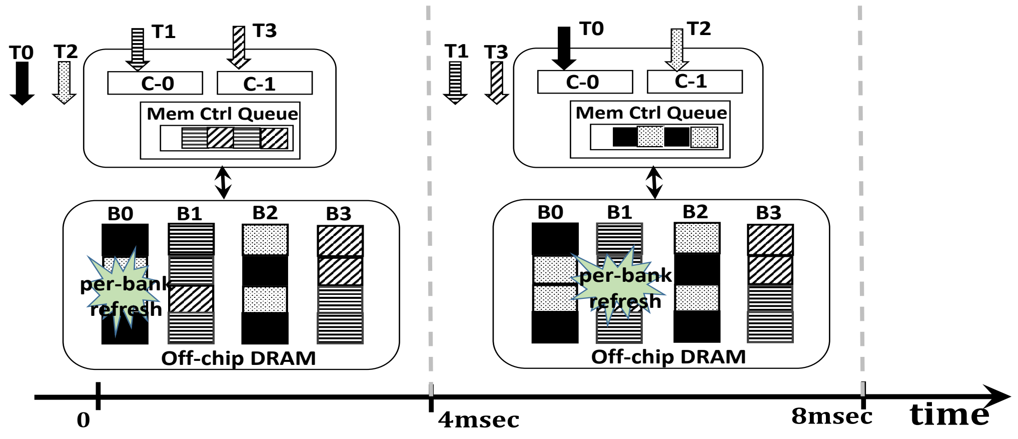


Figure 6.7: Refresh-Aware process scheduler (with soft-partitioning allocator).

6.4.3 Putting It All Together

Figure 6.7 depicts the bigger picture of how our co-design works. As discussed in Section 6.4.1, our proposed per-bank refresh schedule results in Bank B-0 being

Hardware Config	**Cores**	2 cores @ 3.2GHz, Out-of-order, 8-wide issue, ROB: 128 Entries.
	L1I/L1D $	32KB, 4-way associative, Hit latency: 2 cycles
	L2 Cache	1MB per core, 2MB total, 16-way associative, Hit latency: 20 cycles, 64 Byte cache lines.
	Memory	DDR3-1600 [50], 1 channel, 1DIMM/channel, 2 ranks/DIMM, 8 banks/rank, FR-FCFS scheduler, open-row policy, Read/Write Queue Sizes: 64/64, Writes drained in batches [15] [100], Low/High watermarks: 32/54, 4KB DRAM row
	Refresh Config	$tREFI_{ab}$=7.8 μsecs, $tRFC_{ab}$-to-$tRFC_{pb}$ ratio = 2.3 [15] $tRFC_{ab}$ = 530/710/890 nsecs for 16Gb/24Gb/32Gb chips, Rows/bank = 256K/384K/512K for 16Gb/24Gb/32Gb, tREFW=64msec.
OS Config	**Timeslice**	4msec.
	OS Scheduler	CFS (round-robin).
	Baseline Allocator	Buddy Allocator without any memory partitioning.
	Co-design Allocator	Soft-partitioning based memory allocator.

Table 6.1: Evaluated configuration.

refreshed in the first 4msec, bank B-1 in 4-8msec, and so on. Figure 6.7 shows how data for tasks T0-T4 are allocated based on the soft-partitioning discussed in Section 6.4.2.1. The data of tasks T0 and T2 are allocated on banks B0 and B2, while T1 and T3 have their data allocated on banks B-1 and B-3. Since bank B-0 containing data allocated by tasks T-0 and T-2 will be refreshed in the first 4msec, our refresh-aware OS scheduler schedules tasks T1 and T3 on cores C-0 and C-1. After 4 msec, since bank B-1 will be refreshed from 4-8 msec, tasks T0 and T2 will be scheduled by our refresh-aware scheduler, thereby ensuring that none of the on-demand requests from the scheduled tasks are stalled by the refreshes.

6.4.4 Caveats

In a real-life system, there could be varying scenarios where the process scheduling is disrupted by the external factors. Such scenarios include:

- A high priority task enters the system warranting for it to be scheduled for more number of time quantums compared to the other tasks.
- It could be possible that the desired tasks to be scheduled (that do not have data allocated on the bank to be refreshed) are not in runnable queue as they are in other states e.g., sleep state.
- A non-maskable interrupt needs to be serviced immediately by the core.

In all the above scenarios, our refresh-aware scheduler might result in loss of fairness. To address these issues, “fairness_threshold", denoted by η_{thresh} and depicted in line 29 of Algorithm 12, can be used to disable our refresh-aware co-schedule immediately by setting this parameter to 1 or gracefully by setting to some value like 2 or 3. This η_{thresh} parameter can be used by the user to over-ride the refresh-aware scheduling decisions at-will using the sysctl_sched interface present in the Linux kernel.

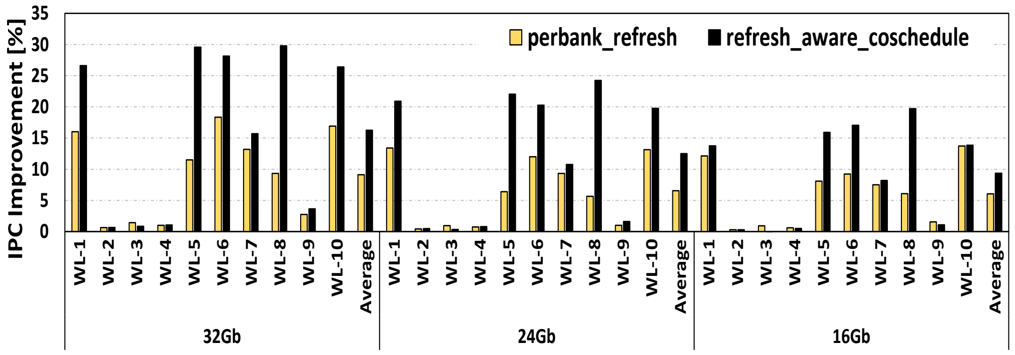


Figure 6.8: IPC improvement results (normalized to all-bank refresh).

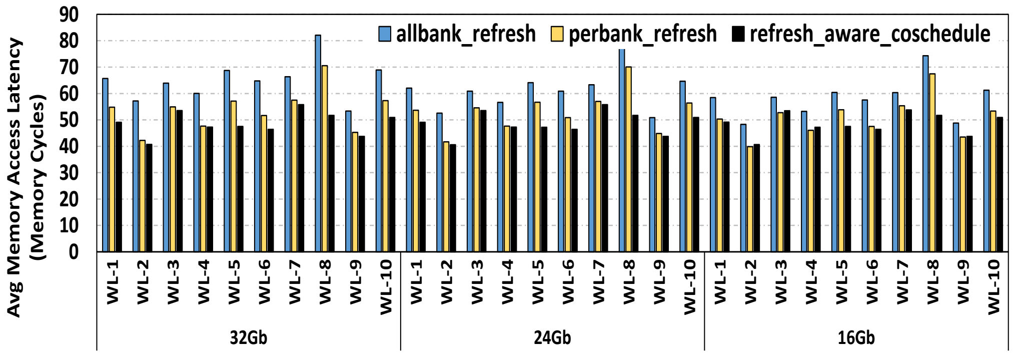


Figure 6.9: Average memory access latency results.

6.5 Evaluation

6.5.1 Experimental Setup

We used a simulation based setup with modified Linux kernel to evaluate our co-design. For the simulation setup, we used the gem5 [101] simulator with the out-of-order CPU model integrated with NVMain [102] for the detailed memory model. The evaluated system configuration is given in Table 7.1, unless otherwise explicitly stated. Our default experiments are evaluated with 4 threads consolidated per core, with a default system executing 8 threads in total on 2 cores (a 1:4 consolidation ratio). As mentioned in Section 2.2.2, by registering a callback to switch_to( ) Linux system call in the gem5 simulator, we observed that each task in our workloads covered in Table 6.2 executes for a time-slice of 4msec. Hence, we use the baseline round-robin schedule of tasks from this callback handler in our gem5 simulator.

We used benchmarks from the SPEC CPU2006 [44] suite with the reference (large) input, STREAM [96] and UA from NAS [97] benchmark suite. We evaluated various multi-programmed workloads shown in Table 6.2, each using a mix of these benchmarks based on their memory intensities. We categorize an application with Misses Per Kilo Instruction (MPKI) higher than 10 as high memory intensive application, denoted by H in the table. Applications with MPKI values between 1 and 10 are categorized as medium, denoted by M, and those with MPKI values less than 1 are categorized as low. As shown in Table 6.2, our workloads are formed such that we cover a large spectrum of the memory intensity so that our performance results reported are representative and are not biased by high memory intensive workloads. For evaluation, we fast-forward applications to get to the region-of-interest after which the workload executes 100 million instructions to warm-up the LLC. We continue the simulation till each task in the workload executes a minimum of 200 million instructions. Once the last task finishes executing its 200 million instructions, we terminate the simulation and dump the statistics. Performance improvements reported in this section are the improvements in *harmonic mean* of the Instructions committed Per Cycle (IPC) of the workload relative to the baseline.[5]

6.5.2 Co-design Results

Figure 6.8 shows the performance improvements of per-bank refresh and our refresh-aware co-design normalized to all-bank refresh for a dual core system with a 1:4 consolidation ratio. Note that, in our baseline, memory is not partitioned across tasks and a task can allocate data in all the 8 banks in a rank. For our co-design experiments, we confine each task to 6 banks within a rank so that not all tasks

[5]Note that in our baseline, each task is executed in a round-robin fashion with a time-slice of 4msec.

	Benchmarks	MPKI Category
WL-1	mcf(8)	H
WL-2	povray(8)	L
WL-3	h264ref(8)	L
WL-4	povray(4), h264ref(4)	L
WL-5	GemsFDTD(8)	M
WL-6	mcf(4), povray(4)	H + L
WL-7	stream(4), h264ref(4)	M + L
WL-8	bwaves(4), h264ref(4)	H + L
WL-9	npb_ua(4), povray(4)	M + L
WL-10	mcf(4), bwaves(2), povray(2)	H + L

Table 6.2: Workloads used in evaluating our co-design in a dual-core system (1:4 consolidation ratio).

have data allocated on all 8 banks in a rank. Confining a task to 6 banks[6] in a dual-core system is the sweet-spot as it gives us good BLP (thereby reducing contention) as well as gives our co-design a flexibility to schedule tasks such that none of the on-demand requests are stalled by refreshes. As can be observed, our co-design scheme gives significant benefits over the all-bank refresh and per-bank refresh. Our co-design on an average improves the performance by 16.2%, compared to all-bank refresh, while it improves the performance by 6.3% over per-bank refresh for 32Gb chips. For 24Gb chip density, our co-design improves the performance by an average of 12.1% over all-bank refresh, and it improves the performance by an average of 5.4% over per-bank refresh. Furthermore, as the refresh overheads become less of a problem for 16Gb chips, our co-design improves the performance by 9.03% and 2.5% over the all-bank and per-bank refresh schemes, respectively. Figure 6.9 shows the corresponding average memory latencies in memory cycles per workload (lower the better in this graphs). As expected, the average memory access latencies are reduced significantly by our co-design as none of the tasks' on-demand requests are stalled by the refreshes.

The workloads WL-2, WL-3 and WL-4 are low memory-intensive, and hence not many of their on-demand requests are stalled by the refreshes in the baseline itself. Consequently, they do not get any improvement in performance from our co-design (as can be noted from Figure 6.8). WL1 as presented in Table 6.2 comprises of mcf applications which has a very high MPKI, compared to the other benchmarks categorized as high. Since our approach confines each task to 6 out of the 8 banks, the tasks executing at the same time contend for bandwidth from the confined banks. As a result, the improvement in performance is significant but still not as

[6]We have experimented with 4 and 2 banks as well. While they improve performance,the improvements are not as high as 6 banks case.

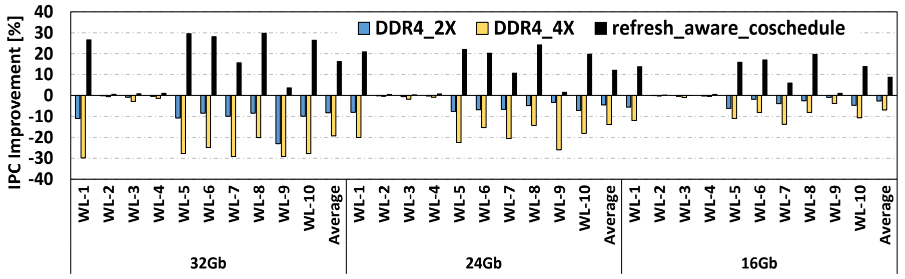


Figure 6.10: Comparison with FGR in DDR4 (normalized to allbank-refresh DDR4-1x mode baseline).

significant as other high MPKI workloads. As can be observed, WL5 comprising of the medium intensive applications like GemsFDTD and WL8 comprising of a mix of the high and low MPKI workloads give very good improvements as there is not much contention for bandwidth in the confined banks. Our co-design gave significant overall energy savings as well compared to the all-bank and per-bank refresh scenarios. We could not present them in the interest of space.

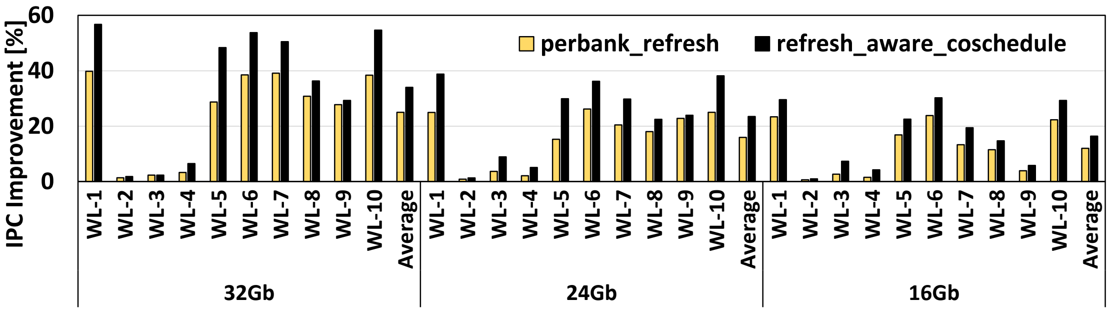


Figure 6.11: Results with the 32msec retention time.

6.5.3 DDR4 Fine Granularity Refresh Results

Figure 6.10 shows how our co-design fares with DDR4-1600. As discussed in Section 2.1.3.2, DDR4 supports 1x, 2x and 4x refresh modes [51]. DDR4 1X mode employs a $tREFI_{ab}$ of 7.8 μsecs, while 2x and 4x modes employ 3.9 μsecs and 1.95 μsecs respectively. While the $tREFI_{ab}$ is halved from 1x to 2x and 2x to 4x modes, $tRFC_{ab}$ for 2x/4x modes is scaled only by a factor of 1.35x/1.63x as observed in [15] [16]. Consequently, DDR4-2x and DDR4-4x modes fare worse than DDR-1x as more number of refresh commands are issued in a given refresh window, thereby causing more number of on-demand request stalls. To mitigate these refresh overheads, Adapative Refresh (AR) [16] dynamically switches between the DDR4-1x and DDR4-4x modes by monitoring the DRAM channel utilization at runtime. We present the comparison results of our co-design with AR in Section 6.5.5. As the entire refresh overheads are masked in our co-design, our co-design performs significantly better on an average compared to DDR4-1x, DDR4-2x and DDR4-4x modes, as can be noted in Figure 6.10 as we schedule tasks in a refresh-aware fashion.

6.5.4 Results with Lower DRAM Retention Time

As covered in Section 2.1.3.2, the DRAM retention time is halved to 32msecs in environments operating beyond 85 deg C. As a result, the DRAM refresh overheads become detrimental to the overall system performance. Using a refresh window, tREFW of 32msec, Figure 6.11 shows the performance improvements acheived by our co-design.[7] As in other performance graphs, all the results are normalized to all-bank refresh baseline. Our co-design refresh improves the performance in such high temperature environments on an average by 34.1% over all-bank refresh and 6.7% over per-bank refresh for 32Gb chips. In 24Gb chips, our co-design improves the performance on an average by 23.4% and 6.3% over the all-bank and per-bank refresh, respectively, while in 16Gb chips, the average performance improvements are 16.4% and 3.9%, respectively, over all-bank and per-bank refresh.

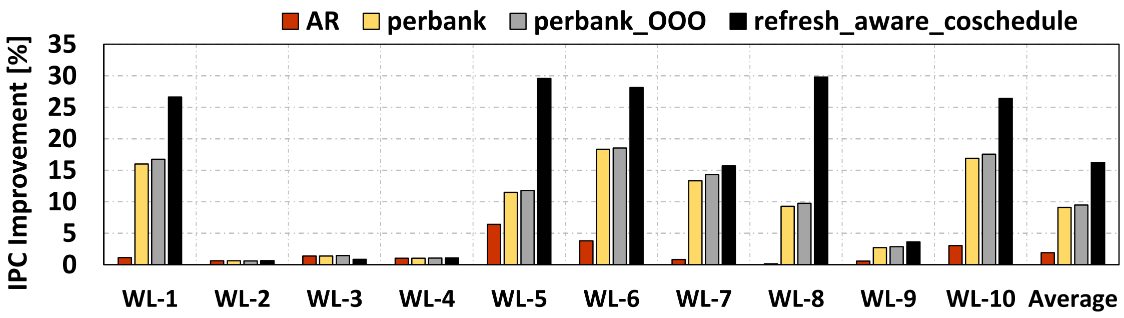


Figure 6.12: Comparison results for 32Gb chips (normalized to all-bank refresh).

[7]Note that we used a 2msec time-slice for 32msec retention time in our experiments, which still falls in typical OS time-slice duration of 1-5msec [24].

6.5.5 Comparison with Previous Proposals

Figure 6.12 shows how our co-design fares over some of the previously proposed hardware-only solutions. Since our mechanism is based on the per-bank refresh, we compare it with the out-of-order (OOO) per-bank refresh proposed by Chang et al [15]. Apart from doing a OOO per-bank refresh, they further propose parallelizing accesses going to the refreshed bank by assuming sub-array support. Since our mechanism does not assume these additional support (modifications) to a DRAM bank, we compare our co-design only with OOO per-bank refresh. In OOO per-bank refresh, while deciding which bank to be refreshed, they look at the transaction queue and decide the target bank as the one with the lowest number of outstanding requests. As can be observed in Figure 6.12, just performing an OOO per-bank refresh does not improve the performance considerably. In our experiments, we observed that this is primarily a timing issue. Even though there are no requests queued to the target bank when deciding which bank to be refreshed, as the refresh operation lasts for several hundred *nanoseconds* ($tRFC_{pb}$), we observed the outstanding requests to the bank being refreshed increased from the point the decision is taken. This is primarily because the data of each task are spread across all the banks. As a result, the average performance improvement brought in by the OOO per-bank refresh is marginal compared to the per-bank refresh but is significant around 9.5% compared to the all-bank refresh for 32Gb chips. Our co-design performs significantly better compared to the OOO per-bank refresh improving the performance on an average by 6.1%. In the interest of space, we could not present the results for other chip densities, but they seem to follow the same trend.

Figure 6.12 also presents results compared to another previously proposed hardware-only solution, Adaptive Refresh (AR) [16]. As discussed in Section 6.6, AR switches between the DDR4-1x and DDR4-4x modes dynamically by monitoring the channel bandwidth utilization. AR is an optimization proposed on top of DDR4 all-bank refresh. As can be noted from Figure 6.12, AR improves the performance by 1.9% on average compared to all-bank refresh but still does not perform as well as the per-bank refresh. Similar observations have also been noted by Chang et al [15]. Compared to AR, our co-design improves the performance on an average by 14.6%.

6.5.6 Sensitivity Results

Figure 6.13 shows the sensitivity results of our co-design with varying number of cores and varying number of tasks per core. In the interest of space, we present the average improvements over all the workloads (and not each workload result) across different chip densities. As can be observed, our co-design consistently fares better than both all-bank and per-bank refresh across various consolidation ratios. Confining to use just 1 DRAM channel, in a dual-core system as the consolidation ratio decreases from 1:4 to 1:2, a task's data can only be allocated on 4 banks per rank, as opposed to 6 banks per rank in the 1:4 consolidation ratio. This is because, assigning more than 6 banks per rank for each task allows only one task to be available to be scheduled on a dual-core system remaining the other core idle, thereby resulting in the under-utilization of cores. Hence, on a dual-core system with the 1:2 consolidation ratio, memory is partitioned such that each task allocates data on 4 banks in a rank, making a total of 8 banks. Consequently, the available BLP is reduced compared to the 1:4 consolidation ratio scenario. However, the 1:2 consolidation ratio still fares better compared to the all-bank and per-bank refresh. Our co-design improves the performance by 14.2%,11.2%,8.9% over all-bank refresh in 32Gb, 24Gb and 16Gb chips, respectively. However, by scaling up the number of DIMMs per channel from 1 to 2, it is possible for each task to allocate data on more number of banks, resulting in improved BLP and reduced contention and ultimately higher performance benefits. As can be observed, our co-design also gives good performance improvements as we increase the number of cores and the corresponding number of tasks per bank over the all-bank and per-bank refresh scenarios as well.

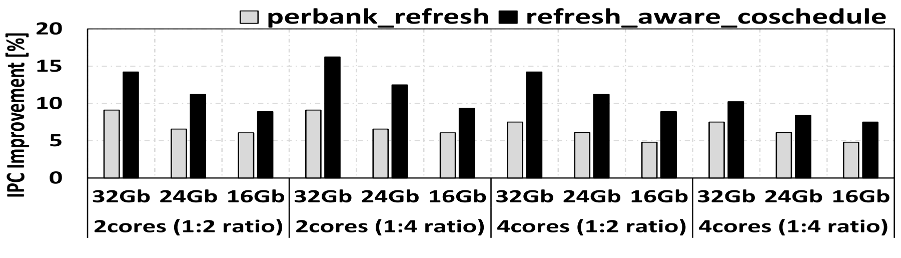


Figure 6.13: Sensitivity results (normalized to all-bank refresh).

6.6 Related Work

Many recent works have proposed hardware and software solutions to mitigate the DRAM refresh overheads. These solutions reduce the DRAM refresh overheads by either skipping unnecessary refreshes or allowing DRAM accesses to proceed in parallel with refreshes.

Multiple previous works have exploited the fact that most of the DRAM cells have high retention times and do not need to be refreshed as often as the small number of weak cells with low retention times. Liu et al. proposed RAIDR [14], a retention-aware refresh technique which enables 75% of the refresh activity to be eliminated. Bhati et al. proposed modifications to the existing auto-refresh functionality in order to enable such refresh skipping [17]. Other software techniques such as Flikker [94] and RAPID [95] take retention times into account while allocating the critical program data and the OS pages, respectively. All these techniques rely on building a retention time profile for the entire DRAM, which requires extensive testing and could incur a substantial runtime overhead. Furthermore, recent work has shown that DRAM cell retention times exhibit large variations with both time and temperature [14] [93], making "retention time profiling" unreliable and difficult to implement.

Other prior work attempts to reduce refresh overheads by scheduling refresh commands in periods of low DRAM activity. Elastic Refresh proposed by Stuecheli et al. [103] postpones up to 8 refresh commands in order to find idle periods when these refresh commands could be scheduled. Similarly, Co-ordinated Refresh [104] attempts to schedule refreshes when DRAM is in the self-refresh mode. While these techniques could save refresh power for low memory intensity workloads, they may not work well for memory-intensive workloads where periods of low memory activity are scarce.

Another approach adopted by prior techniques to reduce refresh overheads is to use finer granularity refresh modes. We already presented how our co-design fares quantitatively relative to Adaptive Refresh [16] and DDR4 2x and 4x modes. Adaptive Refresh (AR) chooses one of the three available refresh modes (1x, 2x and 4x) in DDR4, based on monitoring the runtime DRAM bandwidth utilization. Another technique, refresh pausing [105], aborts refresh commands upon receiving DRAM requests and then resumes them subsequently. However, supporting this functionality requires the memory controller to have intricate vendor specific knowledge of the refresh implementation within the DRAM device.

Finally, some recent works have proposed techniques to overlap memory accesses with refreshes. The per-bank feature in LPDDR3 allows one DRAM bank to be refreshed while other banks can be accessed in parallel. Chang et. al [15] and Zhang et al. [106] have proposed techniques to enable bank-granularity and sub-array granularity refresh commands in order to allow more parallelism between refreshes and requests. These techniques require changes to the DRAM subarray architecture. In comparison, our technique enables parallelism of refreshes and requests by careful hardware-software co-design, without requiring any DRAM modifications. If such DRAM modifications are incorporated into the future DRAMs, we expect our co-design to yield even better performance improvements. This is because, exposing the sub-array structures to the OS can enable soft-partitioning at a sub-array granularity, resulting in reduced contention and increased BLP.

Chapter 7

Dynamically Reconfigurable Memory System

7.1 Introduction

Many client, mobile, and data-center applications have a growing demand for memories with faster access latency, larger capacity and higher bandwidth. With the increasing popularity of many data-intensive applications including high-resolution graphics and machine learning, memory is increasingly becoming a performance bottleneck. Due to the significant speed disparity between CPUs and memory, improvements in memory latency and bandwidth lead to substantial performance improvements in memory-bound applications. This demand can be alleviated using die-stacked or on-die memories (e.g., HBM [3], MC-DRAM [72], HMC [107]). For example, Intel KNL systems use a 16 GB of MC-DRAM [72].

Unfortunately, it is cost-prohibitive to use more expensive memories as a sole memory component in a system. This observation led to developing heterogeneous memory systems[1] that combine a fast memory component (e.g., die-stacked DRAM) with a (typically larger) relatively slow memory components (e.g., DDR3/DDR4, GDDR DRAMs) [4, 108, 109]. Many prior proposals use fast memory in a heterogeneous memory system as a cache [110–119]. Caches provide performance advantages due to spatial and temporal locality where a large fraction of memory references

[1]Note that a heterogeneous memory system can refer to memories differing in technology, volatility, endurance etc. In this work however, heterogeneous memories refer to memories having varying access-times/bandwidth.

access fast memory. However, since caches duplicate data, they reduce the overall OS-visible memory capacity which degrades performance for high memory-footprint applications. This is especially a problem when fast memory constitutes a large fraction of the overall memory capacity, and when a multi-program workload strains memory capacity resources.

To enable the performance improvements of fast memories while avoiding the capacity reduction of using them as caches, Part of Memory (PoM) architectures which expose the stacked DRAM to OS to enhance the overall OS-visible capacity have been proposed [120–123]. PoM architectures could be hardware-managed [120,123] or could be used as an OS-managed NUMA system [121,122]. OS-managed memory systems provide a low-overhead mechanism to achieve performance, but do not react quickly to changing memory demands of running applications. Hardware-managed PoM systems adapt quickly to changing memory demand and therefore outperform OS-managed heterogeneous memory, but this comes at the expense of higher area and power to manage hardware address indirection and large region swaps between fast and slow memories [123]. Due to the necessity of swapping large segments or pages between fast and slow memories, PoM incurs a large power overhead, and could degrade performance when swaps interfere with on-demand memory accesses.

In this work, we propose and evaluate a novel architecture, Chameleon, which attempts to achieve the best of cache and PoM architectures using a hardware-software co-design. Chameleon uses a hardware-managed PoM as the baseline to achieve its capacity advantages. However, we rely on the operating system to inform hardware of any pages that have been allocated or freed using two new instructions: *ISA-Alloc* and *ISA-Free*. Based on this information from the OS, Chameleon attempts to use freed pages in fast memory as a hardware-managed cache, therefore avoiding the expensive page swap operations. Chameleon switches between cache and PoM modes based on the available free-space.

To the best of our knowledge, this work represents the first work that provides a hybrid solution that dynamically adapts separate memory regions to use as a cache or PoM. More specifically, this work makes the following contributions:

1. We demonstrate that different workloads exhibit different memory-footprints over time.
2. We propose a novel architecture, Chameleon, which uses free pages in fast

memory to implement a hardware-managed cache. This architecture provides PoM-like memory capacity with cache-like performance.

3. We propose simple changes in the instruction set architecture to support Chameleon. With two new instructions, the OS can inform the hardware when a page is freed or allocated, allowing Chameleon to use free pages as a cache.
4. We show that Chameleon achieves the best of both PoM and cache architectures, outperforming a PoM baseline by 11.6% and a latency-optimized cache by 24.2%.

7.2 Background and Related Work

7.2.1 Single-Socket Heterogeneous Memory System: NUMA Dichotomy

With the advent of high-bandwidth stacked DRAM memories integrated on the die through a silicon transposer, each socket is itself turning into a NUMA system. This is because, accesses to the on-chip stacked DRAM are faster compared to off-chip memory. Figure 7.1b shows a block-diagram of a heterogeneous system containing both stacked DRAM and off-chip DRAM in the same socket.

7.2.2 NUMA-Aware System-Software Optimizations

7.2.2.1 NUMA-Aware Memory Allocator

Traditional NUMA-Aware Linux and VMware ESXi, by default, cater to the memory allocation requests of a task by allocating the data in the same socket on which the task is currently running to maximize local memory accesses [124, 125]. This is widely referred to as “first-touch” based allocation or local memory allocation policy in Linux. These allocation policies improve performance by minimizing remote memory accesses.

7.2.2.2 Linux Automatic NUMA Balancing (AutoNUMA)

Linux supports an advanced Automatic NUMA balancing mechanism to enhance the locality between the executing task and its corresponding memory [66]. On a

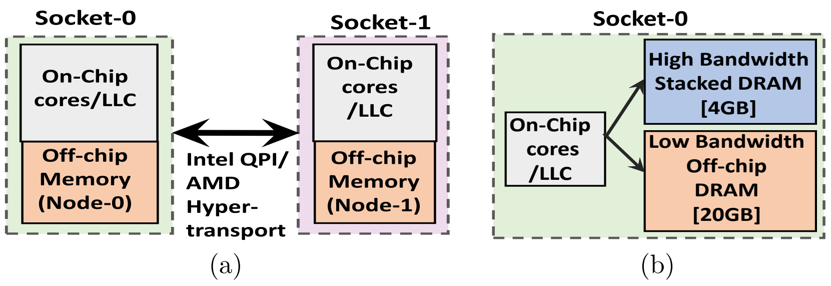


Figure 7.1: (a)Two-socket homogeneous memory system, and (b) Single-socket heterogeneous memory system

multi-socket system, AutoNUMA keeps track of local-to-remote memory accesses by poisoning some set of pages (i.e., invalidating the corresponding page table entries). As a result, a processor load/store to the corresponding page results in a page-fault. In a given epoch, referred to as "numa_balancing_scan_period" in AutoNUMA, Linux calculates the "remote-to-local page-fault ratio". At the end of numa_scan_period epoch, if the remote-to-local page-fault ratio exceeds a threshold (referred to as "numa_period_threshold"), the misplaced pages which caused remote page-faults are migrated from the remote socket's memory to the local socket's memory. Depending on the remote-to-local page-fault ratio, the numa_scan_period is updated dynamically so that the misplaced pages can be migrated quickly to the local socket.

One important issue in AutoNUMA is that memory pages are migrated from the remote to local socket only till there is enough free space available in the local socket's memory. If there is no free space left in local sockets' memory, misplaced page migration fails with "-ENOMEM" error in AutoNUMA. If the remote-to-local fault ratio continues to increase, since AutoNUMA can no longer migrate memory pages to local socket, it migrates the task from a current socket (say socket-0) to socket-1 to minimize remote memory accesses.

7.2.3 Hardware-Managed Heterogeneous Memory

In the context of single-socket heterogeneous memory systems, the hardware-managed techniques for stacked DRAM primarily falls into three categories: (1) cache-based systems and (2) extension to off-chip memory systems, and (3) statically reconfigurable heterogeneous memories.

7.2.3.1 Stacked DRAM as Cache

A large body of recent work has looked at utilizing stacked DRAM as another cache between the last level cache (LLC) and system memory [112–117, 119, 126]. A DRAM cache provides good performance and software transparency, but needs to appropriately organize the cache structure (data and tags). To study the architectural implications of DRAM cache, prior proposals looked at direct-mapped [117], set-associative [116, 119] and fully-associative [115] cache designs. The DRAM cache design in [115] minimizes the cache-tag overheads by employing a novel virtual-to-cache address mapping scheme (cTLB).

7.2.3.2 Stacked DRAM as Part of Memory

Recent work also studied the usage of stacked DRAM as an OS-visible extension to off-chip memory [120, 123, 127–129]. In particular, [127, 128] proposed software-hardware approaches require OS to detect and collect page access information by identifying the first requested pages and the hot pages (FTHP). However, other proposals [120, 123] explored hardware-based redirection by employing a hardware remapping table to ensure high stacked DRAM hit rates. PoM [123] used 2KB segments while CAMEO [120] used 64B segments. Hence, PoM had lower meta-data overhead while CAMEO used less bandwidth.

7.2.3.3 Statically Reconfigurable Heterogeneous Memories

Realizing the importance of stacked DRAM capacity, Intel KNL supports various modes [72] of stacked DRAM (referred to as MC-DRAM) operation. These modes includes entire 100% cache-mode and 100% OS-visible flat (memory) mode. KNL also supports a hybrid-mode where it can be statically split to use 25% of stacked DRAM as cache while 75% is used as OS-visible memory; or to use 50% of MC-DRAM as a cache and 50% as memory.

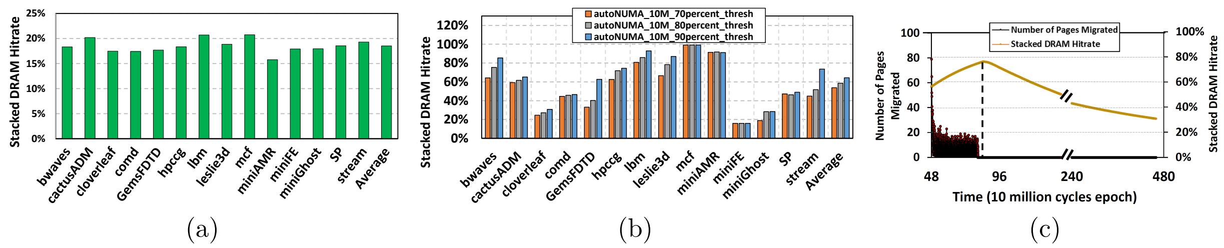


Figure 7.2: (a) NUMA-Aware Allocator (b) AutoNUMA Stacked DRAM hit rates, (c) Cloverleaf AutoNUMA timeline (for 90% threshold).

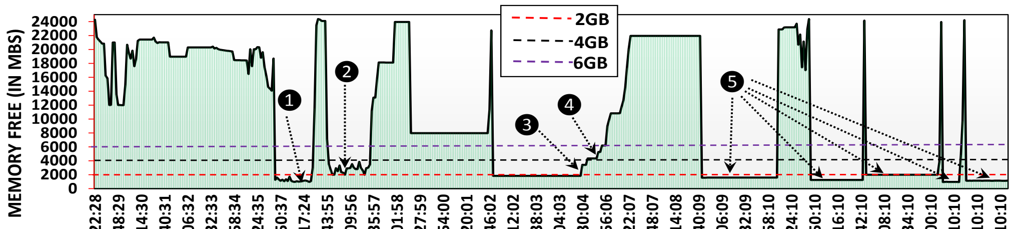


Figure 7.3: Inter-Workload Memory Footprint Variation Over Time (spanning over 2 days).

7.3 Challenges in Architecting Performance-Optimized Heterogeneous Memory

In this section, we describe some OS- and hardware-managed alternatives that could improve the performance of a heterogeneous memory system. We illustrate the role of memory capacity on overall system performance, and show that different workloads exhibit different memory usage over time. We explain some of the challenges facing these architectural alternatives to motivate our Chameleon architecture.

7.3.1 OS-based NUMA Allocation

7.3.1.1 NUMA-Aware Memory Allocator

The Linux NUMA-aware "First-touch" allocator (Section 7.2.2.1, [130]) tries to allocate as many pages as possible in the faster, stacked DRAM to increase the stacked DRAM hit rate. Figure 7.2a shows the stacked DRAM hit rate for high memory footprint workloads in a system containing 4GB stacked DRAM and 20GB off-chip DRAM. The average stacked DRAM hit rate for these high footprint workloads is as low as 18.5%. This low stacked DRAM hit rate is due to two main factors. First, the non-proportional capacity of the stacked DRAM compared to the off-chip DRAM limits the data that a stacked DRAM can accommodate (4GB in our experiments) before it runs out of memory. Second, the OS lacks adequate hot-page prediction mechanisms. Employed page allocation strategies can result in some of the hot-pages getting allocated in the off-chip DRAM, reducing stacked DRAM hit rate.

Our results demonstrate that the NUMA-Aware first-touch memory allocator policy is not optimal for heterogeneous memory systems as it results in severe under-utilization (low hit rate) of the faster stacked DRAM.

7.3.1.2 Linux AutoNUMA

Some of the shortcomings mentioned in Section 7.3.1.1, are successfully handled by Linux AutoNUMA [66]. Figure 7.2b shows the stacked DRAM hit rate for a 4GB stacked + 20GB off-chip DRAM system for different numa_period_threshold values (70%, 80% and 90%). The higher numa_period_threshold yields better stacked DRAM hit rates, on an average of 64.4% as the mis-placed (off-chip DRAM) pages are migrated more rapidly in to the stacked DRAM. Though the average hit rate of AutoNUMA is better compared to the NUMA-Aware allocator, the hit rates are still not desirably high. Specifically, the workloads like Cloverleaf have a cumulative hit rate as low as 30.7% (for 90% threshold). Figure 7.2c, a timeline graph helps us reason about the lower hit rate in the Cloverleaf workload. The primary Y-axis represents the number of pages migrated per epoch, while the secondary Y-axis shows the stacked DRAM hit rate[2]. The X-axis represents the timeline where each epoch is 10 million processor cycles. With time, as the number of mis-placed pages migrated from off-chip DRAM to the stacked DRAM increases, the stacked DRAM hit rate increases, reaching a maximum of around 77.1% at the 81st epoch. However, after epoch 81, as can be observed, the stacked DRAM hit rate reduces gradually from 77.1% to 30.7%. This is because as pages are migrated from the off-chip to the stacked DRAM, the stacked DRAM capacity becomes full. With no free space available to accommodate the mis-placed pages in stacked DRAM, no more pages are migrated and hence the hit rate drops with time ending

[2]Note that there are already pre-allocated memory pages in the stacked DRAM as it is exposed to OS in AutoNUMA.

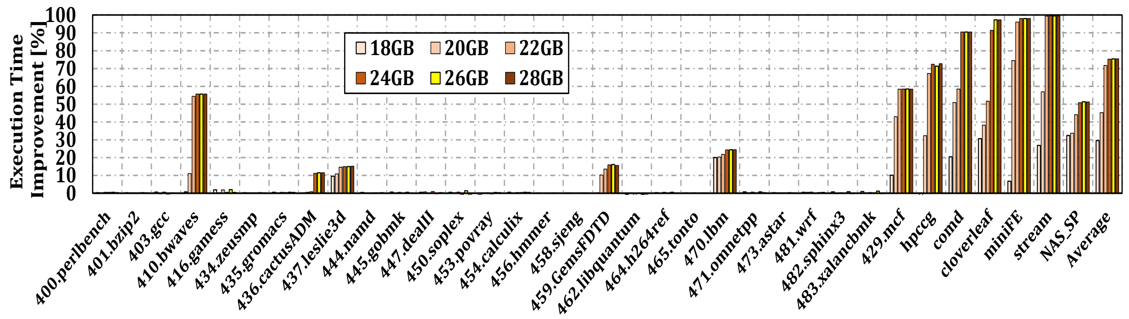


Figure 7.4: Impact of capacity on overall system performance, normalized to a system with 16GB overall capacity.

with 30.7%. The lower stacked DRAM hit rates in AutoNUMA is attributed to the following reasons:

- The single-socket heterogeneous systems contains a non-proportional stacked DRAM capacity unlike the multi-socket systems which contain local memory capacity similar to that of remote memory. Hence, there is a high chance that AutoNUMA can find free memory available in the case of multi-socket system compared to single-socket stacked DRAM system. As a result, AutoNUMA optimized for multi-socket systems does not consider evicting pages from local memory to accommodate misplaced pages to be migrated from the remote memory.
- Even if there is no free space available in the multi-socket system to migrate pages, AutoNUMA increases the local memory accesses by migrating the task to the remote socket. However, this is not feasible in single-socket systems.
- Even in a system with higher free memory space available, since the numa_balancing_scan_period is on the order of milliseconds, the page migration happens at a very coarse – millions of CPU cycles – granularity, as identified in [123].

Summarizing the observations from Sections 7.3.1.1 and 7.3.1.2, the OS-based NUMA optimizations for multi-socket systems do not fare well in single-socket heterogeneous memory systems. They under-utilize the faster stacked DRAM severely, resulting in lower stacked DRAM hit rate. This motivates hardware-managed schemes which can respond to applications' access patterns at a finer – tens to hundreds of CPU cycles – granularity, with flexibility to evict pages from stacked DRAM without necessitating task migrations.

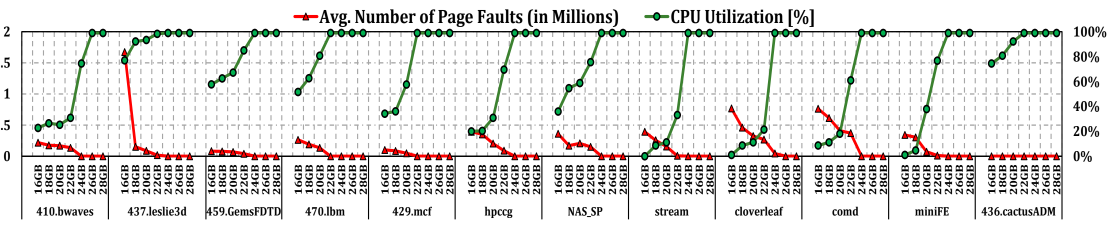


Figure 7.5: Impact of capacity on pagefaults and CPU Utilization for high memory footprint workloads.

7.3.2 Memory Free Space over Time

Figure 7.3 shows the free memory space over time in a system with 24GB overall DRAM capacity. These experiments are conducted on an Intel Xeon CPU E5-2620 (more details about the machine configuration can be found in [131]) running workloads sequentially one after the other spanning over more than 2 days. Each workload in this experiment contains 12 copies of the same application executed in the rate mode [132]. The applications chosen are from SPEC2006 [44], NAS [133], stream [96], and Mantevo [46] suites. The applications used in the workloads for Figure 7.3 can be observed on the X-axis in Figure 7.4. The free memory information is collected using numastat [134] tool in Linux, periodically once every 2 mins.

Figure 7.3 shows that workloads exhibit varying demands on memory over time. The amount of free space in the system can vary from few (including zero) MegaBytes (MBs) to several GigaBytes (GBs).

7.3.3 Impact of Memory Capacity on Performance

As discussed in Section 7.3.2, different workloads are effected differently based on the OS-visible memory capacity. Figure 7.4 shows the ramifications of limiting the overall capacity of a workload as the overall OS-visible capacity is varied from 16GBs to 28GBs, in steps of 2GB, on our Intel Xeon CPU machine [131]. Some applications are agnostic to this variation as their entire memory footprint fits into smaller memory capacity. However, some workloads are sensitive to the overall capacity. The performance improvements reported in Figure 7.4 are *normalized* to a system with 16GB capacity and is calculated as follows:

$$\%Improvement_{xGB} = \frac{((Exec.Time)_{16GB} - (Exec.Time)_{xGB}) * 100}{(Exec.Time)_{16GB}}. \tag{7.1}$$

As the capacity increases from 18GB to 24GB, the average execution time improvement across all the workloads improves from 29.5% to 75.4%, saturating at 75.4% for 26GB and 28GB capacities. The results in Figure 7.5 help us understand the variation of performance with the memory capacity. The results on the primary and secondary Y-axes represent the average number of page-faults (in millions) encountered by the workload and the average CPU utilization of the tasks respectively at the corresponding memory capacity. With the increase in the capacity, the average number of page-faults encountered by OS decreases and the average CPU utilization increases. At lower capacities, most of the time is spent swapping the pages between the DRAM and secondary storage, resulting in poor CPU utilization as the tasks wait in Uninterruptible ("D") state in Linux during the swap. As the capacity increases, page-faults reduce, thereby enabling the tasks to continue in Running ("R") state resulting in 100% CPU utilization.

From Figures 7.4 and 7.5, we can conclude that insufficient memory capacity can result in severe performance degradation. It is also clear (Figure 7.3) that memory demand is a function of the workload running on the system. The design decision of stacked DRAM plays a critical role in overall system performance.

7.3.4 Stacked DRAM as A Cache

While a cache adapts quickly to changing workload behavior, dedicating a memory region as a cache can significantly degrade performance. For example, in Figure 7.3, using 6GB out of the total 24GB capacity as a cache can cause severe performance degradation for workloads operating in regions ❶, ❷, ❸, ❹ and ❺. This is because the overall OS-visible capacity of the system will only be 18GB causing the workloads with higher memory footprints to experience page-faults. A 4GB stacked DRAM cache limits the overall capacity to 20GB, degrading workloads operating in regions marked by ❶, ❷, ❸ and ❺. Even a 2GB stacked DRAM cache would degrade workloads in regions ❶ and ❺. Hence, depending on the capacity of stacked DRAM, the decision of using stacked DRAM as a cache will have adverse-effects on performance/energy of some workloads.

7.3.5 Stacked DRAM as PoM

To avoid memory capacity loss incurred by caches, stacked DRAM can be used as Part of OS-visible Memory (PoM). However, PoM can substantially increase demand for both on-chip and off-chip memory bandwidth. For example, for a stacked DRAM with capacity 6GB out of 24GB, workloads operating in all the regions except ❶ - ❺ will have to swap segments between stacked and off-chip DRAM on every stacked DRAM miss. Such a PoM design would even swap segments from unallocated OS addresses containing invalid data thereby wasting memory bandwidth. For smaller (e.g., 2GB and 4GB) stacked DRAMs, swaps and wasted bandwidth would increase further. Hence, the static decision of designing a stacked DRAM as PoM can result in free-space agnostic swapping, resulting in severe performance degradation, as will be demonstrated in Section 7.6.

7.3.6 Ideal Heterogeneous Memory System

An ideal heterogeneous memory design would *dynamically* provide maximum performance by:

- Reducing page faults for capacity-limited workloads by dynamically operating in the part-of-memory (PoM) mode.
- Optimizing the overhead of swaps for the OS-visible free space by operating in cache mode.
- Optimizing the meta-data (remapping table) overheads.

We propose a novel hardware-software co-design based system which dynamically reconfigures the heterogeneous memory system based on the overall system state. Our proposed system opportunistically converts the OS-visible free space in the system to be used as a hardware-managed cache, while switching to part-of-memory mode for capacity-limited workloads. Based on the free space available in the system, our co-design can operate certain memory regions in the PoM mode while operating the rest in the cache mode. We propose two incarnations of our co-design: (1) CHAMELEON and (2) CHAMELEON-Opt.

7.4 CHAMELEON: Software Support

To communicate the allocated/unallocated[3] physical addresses to hardware, we propose two new processor ISA instructions: *ISA-Alloc* and *ISA-Free*. These instructions are used by OS.

Algorithm 13 OS Memory Allocator Routine

```
struct page * ___alloc_pages(gfp gfp_mask, unsigned int order, struct zonelist zonelist, nodemask_t
nodemask) {
...
page = get_page_from_freelist(gfp_mask, order)  if (page != NULL) goto out  ...
page = alloc_pages_slowpath(gfp_mask, order)  out:
if (page != NULL) {
pageNum = page_to_pfn(page)  pageSize = 0  if (gfp_mask contains (GFP_TRANSHUGE or
GFP_TRANSHUGE_LIGHT) set) {
pageSize = HPAGE_PMD_SIZE  } else {
pageSize = PAGE_SIZE  }
numIterations = pageSize/SegmentSize  for (i: 0, numIterations-1) {

segmentNum = pageNum + (i * segmentSize)  ISA_Alloc(segmentNum)
}
}

return  page;

}
static inline void ISA_Alloc(volatile void *p) {
asm volatile ("isaalloc %0" : "+m"(*(volatile unsigned int *)p));
}
```

Apart from traditional 4KB pages [135], the OS employs transparent huge-pages (THPs) [136] and giant pages [137], to reduce the page-table overheads for workloads with huge memory footprint. Hence, in Chameleon, depending on the granularity of a segment in a segment-group and depending on the granularity of the page being allocated/unallocated, each call to the OS memory allocator/reclamation (free) routines can result in multiple correspondng ISA-Alloc/ISA-Free invocations. In a Chameleon system, Algorithms 13 and 14 present the memory allocator and free routines in OS instrumented with ISA-Alloc and ISA-Free invocations.

As can be observed in line 12 of Algorithm 13, the “gfp_mask" flag in Linux contains the necessary bits to identify the corresponding granularity of allocation. GFP_TRANSHUGE and GFP_TRANSHUGE_LIGHT flags represent the THP allocation requests in Linux and help us detect the granularity of allocation. However, while reclaiming the free-space, since gfp_mask flag is not passed around, the “order" used by Linux can aid in detecting the granularity of the page being freed.

[3]We use the terms unallocation, reclamation and freed synonymously in the rest of the work.

The order is formally defined in [138] and equals $log_2(pageSize)$. The ISA-Free invocation in Linux can be observed in line 19 of Algorithm 14.

Algorithm 14 OS Reclamation Routine

```
static inline void ___free_one_page(struct page * page, unsigned long pfn, struct zone * zone, unsigned
int order, int migratetype) {
pageNum = page_to_pfn(page)
...
list_add(&page–>lru, &zone–>free_area[order].free_list[migratetype]);
out:
zone–>free_area[order].nr_free++;

if(page != NULL) {
pageNum =page_to_pfn(page)
pageSize = 0

if (order == HPAGE_PMD_ORDER) {
pageSize = HPAGE_PMD_SIZE } else {
pageSize = PAGE_SIZE }
numIterations = pageSize/segmentSize for(i: 0, numIterations-1) {
segmentNum = pageNum + (i * segmentSize) ISA_Free(segmentNum)
} }
return 0;
}
static inline void ISA_Free(volatile void *p) {
asm volatile ("isafree %0" : "+m"(*(volatile unsigned int *)p));
}
```

The “segmentSize" in Algorithms 13 and 14 refers to the segment size employed by Chameleon. The various segment granularities supported by the hardware in Chameleon can be easily detected by the OS during boot time. Based on the segment and the allocation granularities, the number of ISA-Alloc and ISA-Free invocations vary. For a 2MB THP allocation/reclamation and for a 2KB segment employed in PoM [123], ISA-Alloc/ISA-Free is invoked 1024 times. However, for a 64-byte cache-line segment employed by CAMEO [120], ISA-Alloc/ISA-Free needs to be invoked 32,768 times. By communicating the allocated/unallocated physical addresses to hardware, hardware can make informed decisions on the operating mode of corresponding memory region.

7.5 CHAMELEON: Hardware Support

Based on the communication from the OS to hardware via the ISA-Alloc and ISA-Free instructions, hardware dynamically reconfigures the heterogeneous memory such that certain sections of the overall-memory will operate in PoM-mode while the others in cache-mode.

As covered in Section 7.2.3.2, Sim et al in [123] proposed a hardware remapping based PoM management scheme. In their design, stacked DRAM is exposed to the OS and the segments from off-chip DRAM and swapped with a segment in stacked DRAM. Each segment is 2KB in granularity in their approach. To reduce the remapping meta-data overheads, Sim et al. employed a Segment-Restricted Remapping technique in [123]. In this technique only the segments with in a "Segment Group" can be swapped with one another. Each Segment Restricted Remapping Table entry in their approach employs a "Shared counter" which aids in swapping the most-frequently used off-chip segment with the corresponding stacked DRAM segment. In summary, combined with coarse 2KB segment granularity and the Segment Restricted Remapping, their technique minimizes the meta-data overheads while successfully mapping the most-frequently accessed segments in stacked DRAM, allowing higher stacked DRAM hit rates. In our Chameleon co-design, we augment the Segment-Restricted Remapping Table (SRRT) employed in [123][4], as shown in Figure 7.6, with additional hardware structures.

The augmented data-structures are shown in red-color (pattern) in Figure 7.6. Apart from the remapping tag-bits and the shared counter in [123], each SRRT entry contains additional Alloc Bit-Vector (ABV), the mode-bit, and the dirty-bit.

In a segment-group, the Alloc Bit-Vector (ABV) signifies whether the corresponding segments have been allocated by the OS (communicated via ISA-Alloc/ISA-Free) or not. If a segment is allocated, the corresponding bit in the ABV is set to 1; else, it is set to 0 (indicating the segment is free). The number of entries in the ABV is equal to the number of segments per segment-group, which is in turn effected by the capacity-ratio between the stacked and off-chip memories. When the system is initially booted, all the bits in the ABV are set to 0, and the bits will be set to 1 as and when the ISA-Alloc is invoked by the OS allocation routine. The mode-bit signifies the operating mode of the segment-group. The mode-bit is set to 0 if the segment-group is operating in the part-of-memory (PoM) mode, and is set to 1 if it is operating in cache-mode. Further details involved in the transitions between PoM- and cache-modes are covered in later part of this section. If a segment-group is in the cache-mode, the dirty-bit in the SRRT indicates if the segment currently

[4]Note that as CAMEO [120] employs a finer-granular 64-byte segments. They employ a huge LLT to track the remapped cache-lines. Though, our Chameleon can augment the CAMEO design, as explained in Section 7.4, the number of segment-groups impacted by an ISA-Alloc or ISA-Free is quite high in CAMEO compared to PoM [123].

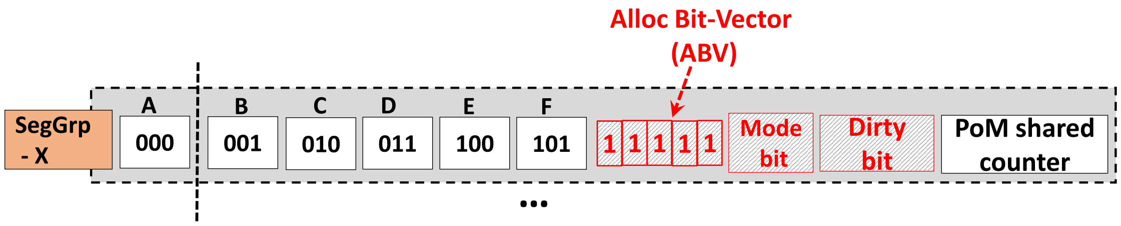


Figure 7.6: Segment Restricted Remapping Table (SRRT) Entry used in our Chameleon co-design.

residing in stacked DRAM is modified or not. As a result, the dirty-bit indicates if the segment needs to be written-back to off-chip memory or not during eviction from stacked DRAM. If the segment is modified, the dirty-bit is set to 1; else, it is set to 0. If the segment-group is operating in PoM-mode, the dirty-bit is set to a don't care “XX" state.

Having understood the SRRT in Chameleon, let us now look at how the hardware dynamically re-configures the heterogeneous memory system to switch between the PoM- and cache-modes based on the allocated/unallocated physical addresses communicated by the OS. At a very high-level, an ISA-Alloc instruction can transition segment-group(s) from the cache-mode to PoM-mode, while an ISA-Free instruction can transition the segment-group operating in the PoM-mode to the cache-mode. However, not all the ISA-Alloc and ISA-Free instructions for the physical addresses in a segment-group will trigger the transitions. This is because, the overall state of a segment-group, i.e., the number of allocated/unallocated segments governs whether an ISA-Alloc or ISA-Free instruction will trigger a transition or not.

Before looking into the scenarios which trigger the transitions, it will help to understand the notation in SRRT shown in Figure 7.6. The segments- A, B, C, D, E and F shown in the SRRT represent the actual physical segments that are part of a segment-group represented by SegGrp-X in the figure. For a 4GB stacked DRAM, a segment from stacked DRAM will have physical address in the range [0, 0xFFFFFFFF][5] , while off-chip segments will belong to range [0x100000000, 0x5FFFFFFFF][5] for a 20GB off-chip DRAM. The tag-bits in Figure 7.6 signify where a corresponding physical address is remapped to (or cached at) in a segment-group.

[5]Aligned at the 2KB segment boundary

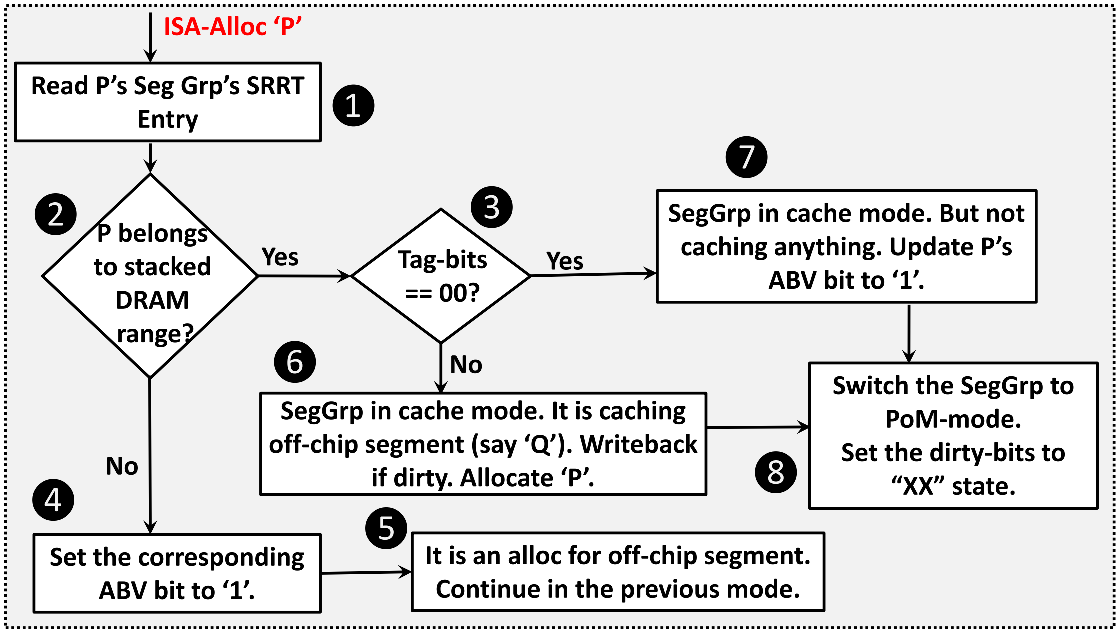

ISA-Alloc 'P'
Read P's Seg Grp's SRRT Entry
1
2
P belongs to stacked DRAM range?
Yes
3
Tag-bits == 00?
Yes
7
SegGrp in cache mode. But not caching anything. Update P's ABV bit to '1'.
No
6
SegGrp in cache mode. It is caching off-chip segment (say 'Q'). Writeback if dirty. Allocate 'P'.
8
Switch the SegGrp to PoM-mode. Set the dirty-bits to "XX" state.
No
4
Set the corresponding ABV bit to '1'.
5
It is an alloc for off-chip segment. Continue in the previous mode.

Figure 7.7: Chameleon ISA-Alloc Transition Flow-chart.

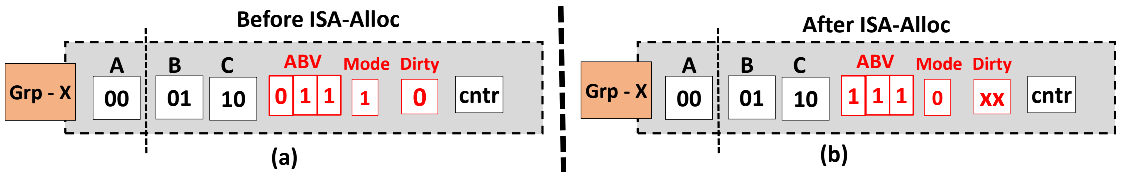


Figure 7.8: Chameleon ISA-Alloc Transition (Example).

7.5.1 Chameleon Design

In our basic Chameleon co-design, OS-visible free-space available in the stacked DRAM can only be leveraged as a cache. As a result, the transitions from PoM- to cache-mode and vice-versa in Chameleon is only triggered by ISA-Alloc/ISA-Free for addresses belonging to the stacked DRAM address range.

7.5.1.1 ISA-Alloc Transitions

Figure 7.7 shows the flow-chart for ISA-Alloc transitions in Chameleon. If the ISA-Alloc is for a off-chip DRAM physical address, as can be observed in the flow ❶ → ❷ → ❹ → ❺ in Figure 7.7, the segment-group continues to operate in the previous mode without any transitions. However, if the ISA-Alloc is for a stacked DRAM physical address, the segment-group will be operating in the cache-mode before encountering this ISA-Alloc. However, by the time ISA-Alloc is encountered, the segment in stacked DRAM could be caching an off-cip segment or not.

Figure 7.8(a) represents a scenario where ISA-Alloc is encountered when none of the off-chip segments is cached in the stacked DRAM (tag-bits representing: 00). In such a scenario, ISA-Alloc will follow the flow: ❶ $\rightarrow$ ❷ $\rightarrow$ ❸ $\rightarrow$ ❼ $\rightarrow$ ❽. From Figure 7.8(a), since segment-A is not allocated, the ABV for segment-A is still 0 while the segment-group (Grp-X) is operating in cache mode (mode-bit: 1). As shown in Figure 7.8(b), after ISA-Alloc, the ABV for segment-A is set to 1 and the segment-group transitions to PoM-mode.

For the other scenario, where an off-chip segment is cached in the stacked DRAM indicated by the tag-bits. If the dirty-bit is set, the corresponding segment is written back to the original segment, else the tag-bits can be simply over-written indicating that the original stacked DRAM segment itself resides in stacked DRAM. This is represented by the flow: ❶ $\rightarrow$ ❷ $\rightarrow$ ❸ $\rightarrow$ ❻ $\rightarrow$ ❽ in flow-chart in Figure 7.7. Finally, the ABV for the stacked DRAM segment is set to ’1’ indicating that it is allocated.

7.5.1.2 ISA-Free Transitions

Figure 7.9 shows the flow-chart for ISA-Free transitions in our Chameleon design. As discussed before, for an ISA-Free to off-chip physical address, there is no transition in segment-group modes; just the corresponding ABV bit is set to ’0’.

If an ISA-Free is encountered for stacked DRAM physical address, prior to encountering ISA-Free, the segment-group will be operating in PoM-mode. Similar to the discussion in ISA-Alloc transitions, there are two scenarios for ISA-Free depending on whether the segment to be freed is remapped with something else or not. If the tag-bits indicate the segment to be freed is not re-mapped, the corresponding ABV bit is set to ’0’ and the segment-group transitions to cache-mode following the flow: ❶ $\rightarrow$ ❷ $\rightarrow$ ❸ $\rightarrow$ ❼ $\rightarrow$ ❽. The tag-bits are set to ’00’ indicating none of the segments are cached in stacked DRAM.

If the segment to be freed is currently not in stacked DRAM as depicted in Figure 7.10(a). As shown in the figure, the original stacked DRAM segment ’A’ is remapped to off-chip DRAM segment ’C’, while ’C’ is itself is remapped to ’B’. This can happen with the following re-mappings. Initially the state is: A, B and C, with A in stacked DRAM and B, C in off-chip DRAM. If segment-C is accessed more frequently, based on the fast-swaps implemented in [123], C will be swapped with

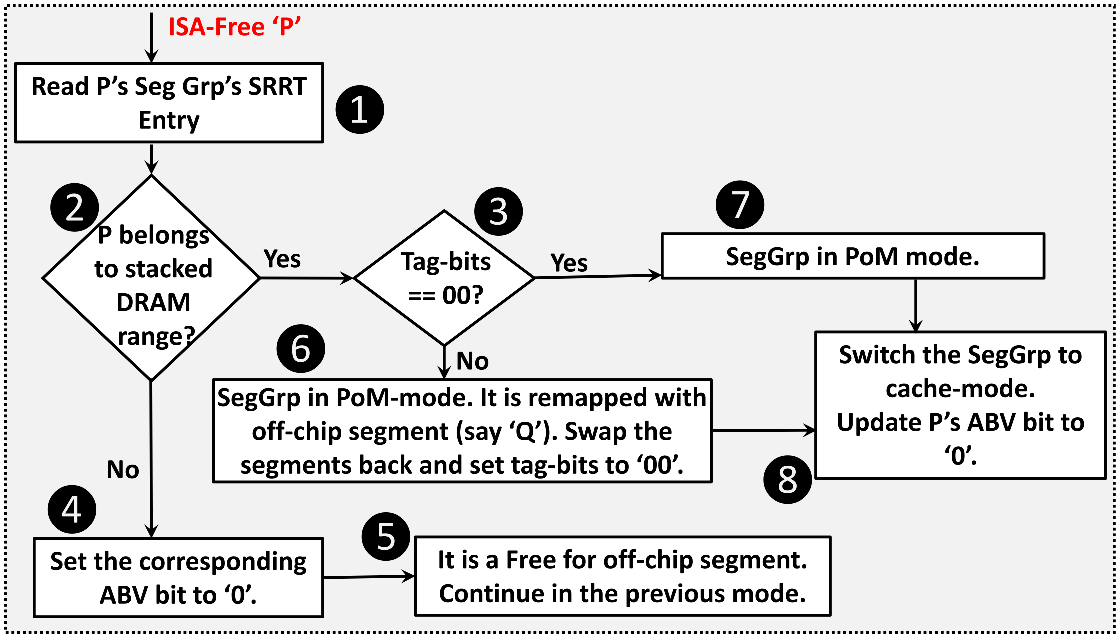

ISA-Free 'P'
Read P's Seg Grp's SRRT Entry
1
2
P belongs to stacked DRAM range?
Yes
3
Tag-bits == 00?
Yes
7
SegGrp in PoM mode.
No
6
SegGrp in PoM-mode. It is remapped with off-chip segment (say 'Q'). Swap the segments back and set tag-bits to '00'.
Switch the SegGrp to cache-mode. Update P's ABV bit to '0'.
8
No
4
Set the corresponding ABV bit to '0'.
5
It is a Free for off-chip segment. Continue in the previous mode.

Figure 7.9: Chameleon ISA-Free Transition Flow-chart.

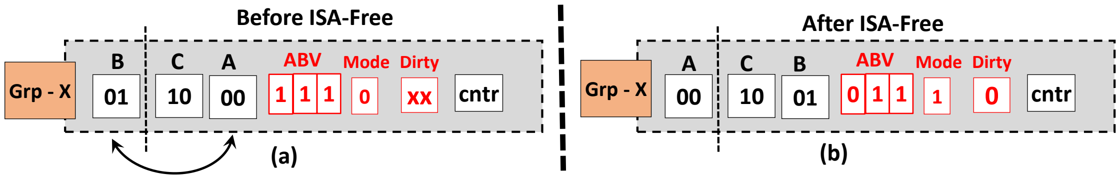


Figure 7.10: Chameleon ISA-Free Transition (Example).

A making the state: C, B and A. This ensures that the most-frequently accessed segment-C resides in stacked DRAM. After such a remapping, in the next program phase, segment-B is accessed more frequently, resulting in swapping of segments-C and B, arriving at a state in Figure 7.10(a) . Now, if ISA-Free happens for segment-A which is the original stacked DRAM segment, the segment-A needs to be swapped with the current segment-B in stacked DRAM before it is freed. Finally, after swapping, as shown in Figure 7.10(b), the segment-A's ABV is set to '0' and the segment-group transitions to cache-mode from PoM-mode following the flow: ❶ → ❷ → ❸ → ❻ → ❽.

7.5.2 Optimized Chameleon (Chameleon-Opt) Design

The Chameleon design discussed in Section 7.5.1 can only leverage the OS-visible free-space in stacked DRAM to be used as a cache even though there are available free segments in the off-chip memory. Consequently, Chameleon is not optimal in terms of leveraging the total available free-space in the system. To overcome these limitations, we present an optimized co-design,Chameleon-Opt, which can pro-actively remap segments in the stacked DRAM to off-chip memory. Such a design can convert the free-space available in both the stacked DRAM and off-chip DRAM to be used as a cache. This optimized design outperforms the Chameleon design, as will be demonstrated in Section 7.6.

7.5.2.1 ISA-Alloc Transitions

Figure 7.11 shows the flow-chart for an ISA-Alloc instruction in Chameleon-Opt. The ABV bits play a crucial role in Chameleon-Opt as they signify when to switch the segment-group from one mode to another. At a very high-level, in Chameleon-Opt, a segment-group remains in cache-mode as long as one of its ABV bits is 0, and it switches to the PoM-mode when all the ABV bits are 1. The test-condition in ❿ in Figure 7.11 represents this check. Hence, unlike Chameleon, a segment-group in Chameleon-Opt continues to stay in the cache-mode even if it's corresponding stacked DRAM address has been allocated by the OS. Similarly, it switches to the cache-mode even if an off-chip physical address has been unallocated (while stacked DRAM segment still remains allocated). This is possible because Chameleon-Opt pro-actively remaps the current segment residing in stacked DRAM to make the OS-visible free-space available in either of the stacked or off-chip DRAM to be available as a free-space in stacked DRAM to leverage as cache.

Unlike Chameleon, in Chameleon-Opt, the segment-group which encounters an ISA-Alloc would already be operating in cache-mode. This is because the very reason that ISA-Alloc is encountered infers that there is at least one OS-visible free segment in the segment-group. As a result, in Chameleon-Opt, the segment-group prior to executing ISA-Alloc always operates in cache-mode. The actions triggered for ISA-Alloc depends on the segment currently residing in the stacked DRAM as well as whether the ISA-Alloc is for a stacked DRAM physical address or not. Following flow: ❶ → ❷ → ❸ → ❹ → ❺ → ❻ in Figure 7.11, as the segment-group operates in cache-mode and there is no segment cached in stacked DRAM (as the tag-bits match the segment being allocated). Therefore, the ISA-Alloc will directly set the ISA-Alloc'ed segments' ABV to '1' and transitions the segment-group to PoM-mode. This is because condition-check at ❹ confirmed that there is no other

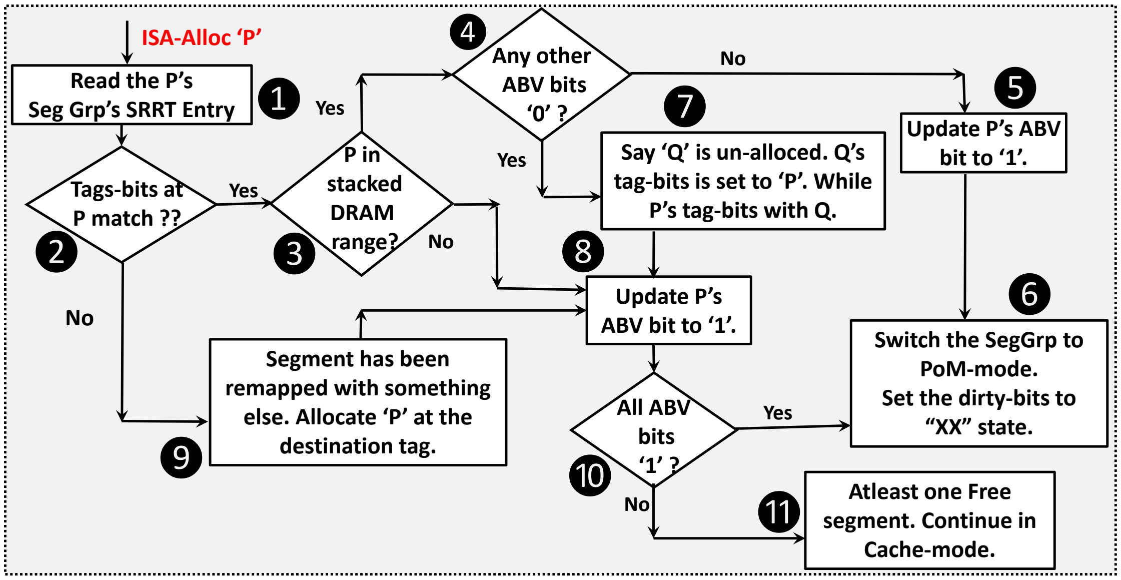

ISA-Alloc 'P'
Read the P's Seg Grp's SRRT Entry
1
Tags-bits at P match ??
2
Yes
No
P in stacked DRAM range?
3
Yes
No
Any other ABV bits '0' ?
4
No
Yes
Update P's ABV bit to '1'.
5
Switch the SegGrp to PoM-mode. Set the dirty-bits to "XX" state.
6
Say 'Q' is un-alloced. Q's tag-bits is set to 'P'. While P's tag-bits with Q.
7
Update P's ABV bit to '1'.
8
Segment has been remapped with something else. Allocate 'P' at the destination tag.
9
All ABV bits '1' ?
10
Yes
No
Atleast one Free segment. Continue in Cache-mode.
11

Figure 7.11: Chameleon-Opt ISA-Alloc Transition Flow-chart.

OS-visible free segments in the segment-group.

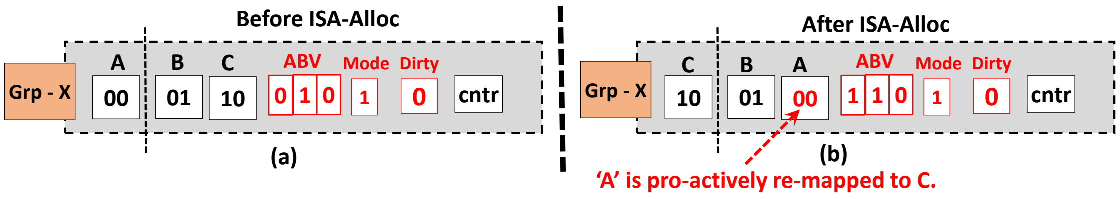


Figure 7.12: Chameleon-Opt ISA-Alloc Transition (Example-1).

Figure 7.12(a) shows one possible scenario for a state prior to executing ISA-Alloc instruction following flow: ❶ → ❷ → ❸ → ❹ → ❼ → ❽ → ❻ of the flow-chart in Figure 7.11. As can be observed, segments- A and C are not allocated, while B is allocated, represented by ABV bits: 0 1 0. And, currently the segment-group is operating in cache-mode (mode-bit: 1). As the stacked DRAM tag-bits match for segment-A, no other segment is currently cached in stacked DRAM. For ISA-Alloc to segment-A, in Chameleon, segment-A is allocated in stacked DRAM (setting it's ABV bit to 1) and the segment-group would have transitioned to PoM-mode. However, in Chameleon-Opt, as depicted in Figure 7.12(b), segment-A is pro-actively remapped to Segment-C in off-chip DRAM. Hence, tag-bits in C is set to "00" while the stacked DRAM segment is set to that of "10" aka segment-C. Since segment-C is never allocated, it will never result in a stacked DRAM hit allowing for either of segments- B or C to be cached. As a result, in Chameleon-Opt, the segment-group still operates in cache-mode unlike Chameleon. This clearly demonstrates how Chameleon-Opt retains free-space in stacked DRAM to be used as a cache.

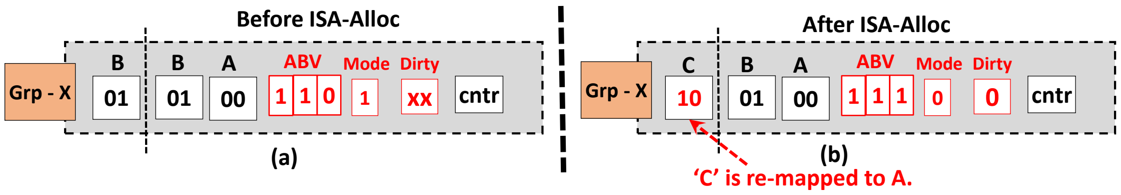


Figure 7.13: Chameleon-Opt ISA-Alloc Transition (Example-2).

Figure 7.13(a) shows a possible scenario for a state prior to executing flow: ❶ → ❷ → ❾ → ❽ → ❿ → ❻. As can be observed, following pro-active remapping explained in Figure 7.12(b), segment-A is remapped to off-chip memory and stacked DRAM caches an off-chip segment B in stacked DRAM as depicted in Figure 7.13(a). Since the segment-group is in cache-mode, the ISA-Alloc'ed segment-C will be allocated in stacked DRAM, changing the corresponding tag-bits to '10' as shown in Figure 7.13(b). As all the ABV bits are '1', the segment-group transitions to PoM-Mode. While in flow ❶ → ❷ → ❾ → ❽ → ❿ → ⓫, since there are more unallocated segments, segment-group continues to operate in cache-mode.

The other flows in Figure 7.11 are pretty straight-forward. For example, flow: ❶ → ❷ → ❸ → ❽ → ❿ → ❻ tackles a scenario where the ISA-Alloc to off-chip address which is not remapped with any other segments with all the other segments allocated by OS. In this case, the segment is allocated at the original off-chip address and the segment-group transitions to PoM-mode. While in flow ❶ → ❷ → ❸ → ❽ → ❿ → ⓫, since there are more unallocated segments, segment-group continues to operate in cache-mode.

7.5.2.2 ISA-Free Transitions

The ISA-Free instruction has a little complicated flow-chart in Chameleon-Opt design as can be observed in Figure 7.14. This is because apart from the free-space available in stacked DRAM, Chameleon-Opt pro-actively converts the free-space available in off-chip DRAM to be visible as free-space in stacked DRAM to be leveraged as cache. The pro-active free-space creation in Chameleon-Opt can be

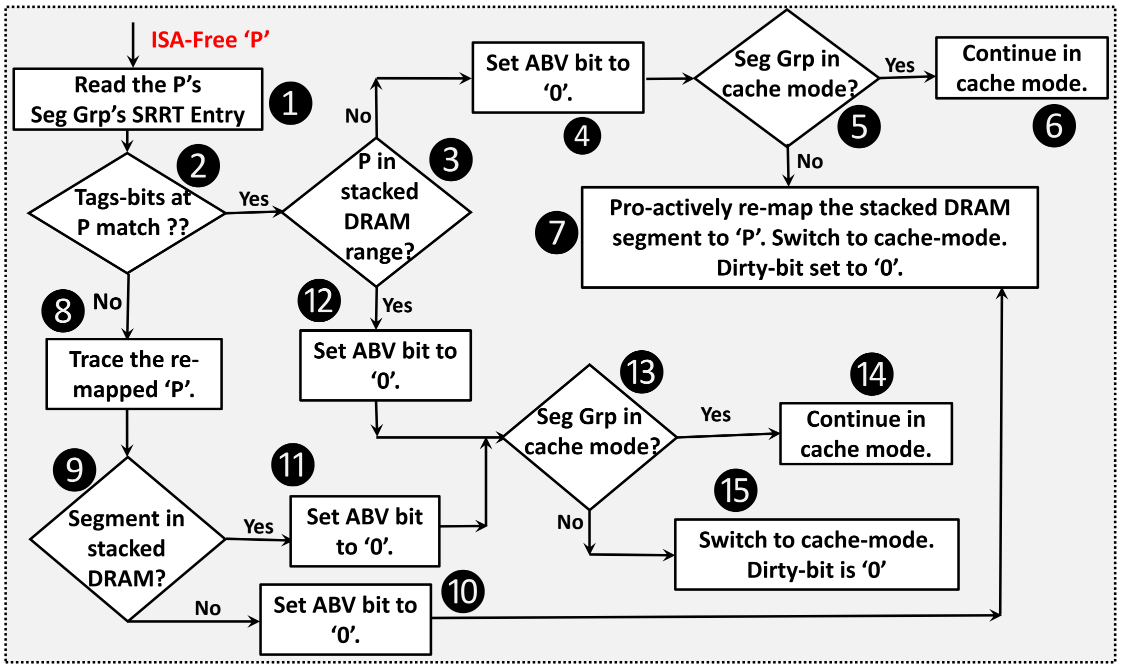


Figure 7.14: Chameleon-Opt ISA-Free Transition Flow-chart.

observed in the flow: ❶ → ❷ → ❸ → ❹ → ❺ → ❼. Figure 7.15 presents a scenario which demonstrates this flow before and after ISA-Free for an off-chip segment. In Figure 7.15(a), it can be observed that the segment-group is operating in PoM-mode as all the segments are allocated by the OS. As the ISA-Free for off-chip segment-C is encountered, segment-C 's ABV bit is set to '0'. However, the stacked DRAM segment, segment-A is remapped to C before switching to the cache-mode as can be observed in Figure 7.15(b).

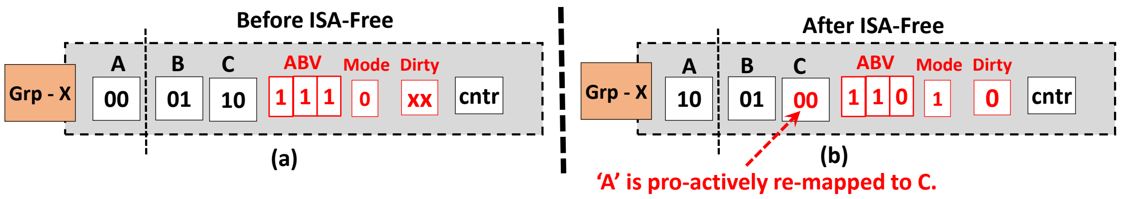


Figure 7.15: Chameleon-Opt ISA-Free Transition (Example).

The other flow, ❶ → ❷ → ❸ → ❹ → ❺ → ❻, represents a scenario similar to the one depicted in Figure 7.15(a), except that the segment-group is already in the cache-mode as there are more free-segments in the segment-group. As a result, the segment-group continues to operate in cache-mode, just the corresponding ABV bit is set to '0'. The flows ❶ → ❷ → ❸ → ⓬ → ⓭ → ⓮ and ❶ → ❷ →

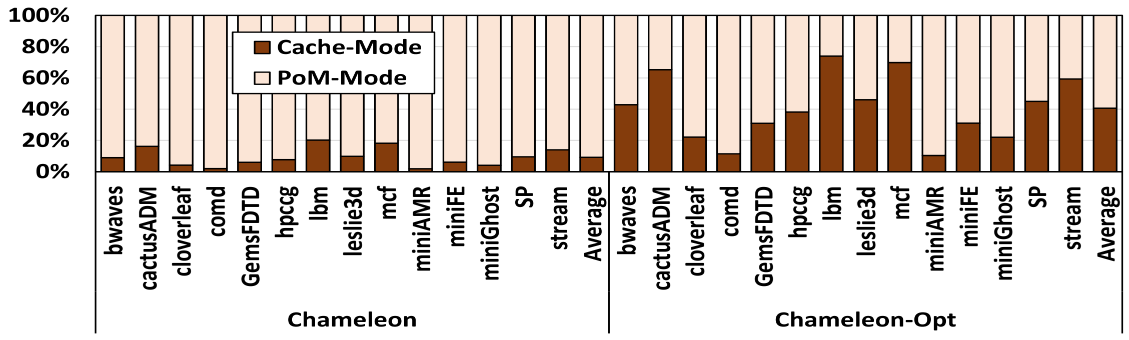


Figure 7.16: PoM to Cache mode segment group distribution.

❸ → ⓬ → ⓭ → ⓯ represent scenarios, where the ISA-Free is encountered for stacked DRAM addresses that are neither caching nor remapped with any off-chip segments, respectively. ISA-Free transitions the segment-group to cache-mode if it was previously operating in PoM-mode. The various other flows in Figure 7.14 correspond to cases where if segment being freed is re-mapped with other segments or not and are easy to follow.

To summarize, as demonstrated in Figures 7.12(b) and 7.15(b), Chameleon-Opt can opportunistically retain/create more free-space to be used as a cache compared to Chameleon.

7.6 Evaluation

7.6.1 Experimental Setup

We evaluated our proposals using GEM5 simulator [101] executing a modified Linux kernel. ISA-Alloc/ISA-Free are invoked in the OS memory allocator/reclamation code using the pseudo-instruction support in GEM5 [139]. Our stacked and off-chip DRAM models are based on the memory controller support in GEM5. Table 7.1 summarizes our simulated configuration. We simulated applications from various suites discussed in Section 7.3.2 whose characteristics are presented in Table 7.2. Our workloads are fast-forwarded to the region of interest and caches are warmed-up. Our simulations execute a minimum of 500 million instructions per application. With 12 copies, we simulate a minimum of 6 billion (500M*12) total instructions. All the reported results are based on 4GB stacked and 20GB off-chip DRAM, unless otherwise stated. In all the performance results reported in Section 7.6.2, the performance reported is the *geometric mean* of Instructions committed Per

Cores	12 @ 3.6GHz (each), ALPHA ISA, out-of-order
L1(I/D)	32KB, 4-way associative, 64B cacheline
L2 Cache	256KB (private), 8-way associative, 64B cacheline
L3 Cache	12MB (shared), 16-way associative, MESI, 64B cacheline
Stacked DRAM	Bus Frequency: 1.6GHz (DDR 3.2GHz), Bus Width: 128 bits/channel, Capacity: 4GB, 2 channels, 2 ranks/channel, 8 banks/rank, 2KB row buffer, tCAS-tRCD-tRP-tRAS: 11-11-11-28, tRFC: 138 nsecs
Off-chip DRAM	Bus Frequency: 800MHz (DDR 1.6GHz), Bus Width: 64 bits/channel, Capacity: 20GB, 2 channels, 2 ranks/channel, 8 banks/rank, 2KB row buffer, tCAS-tRCD-tRP-tRAS: 11-11-11-28, tRFC: 530 nsecs

Table 7.1: Simulated Configuration.

Suite	**WL**	**LLC-MPKI**	**MF**	**Suite**	**WL**	**LLC-MPKI**	**MF**
SPEC2006	bwaves	12.91	21.86	Mantevo	cloverleaf	30.33	23.01
	lbm	29.55	19.17		comd	0.71	23.52
	cactusADM	2.03	20.12		miniAMR	1.44	22.40
	leslie3d	12.18	21.65		hpccg	7.81	22.15
	mcf	59.804	19.65		miniFE	0.48	22.55
	GemsFDTD	20.783	22.56		miniGhost	0.19	20.68
NAS	SP	0.87	21.72	Stream	Stream	35.77	21.66

Table 7.2: Workload Characteristics (MF: Memory Footprint, WL: Workload, MPKI: Misses Per Kilo Instructions)

Cycle (IPC) of all the benchmarks in a workload *normalized* with respect to the corresponding baselines.

7.6.2 Results

Figure 7.16 shows the breakdown of the percentage of segment groups operating in cache mode and PoM mode in Chameleon and Chameleon-Opt designs. On average, 9.2% of the segment groups operate in the cache mode in Chameleon compared to 40.6% in Chameleon-Opt. This is because Chameleon-Opt leverages the free space available in the off-chip DRAM to be used as cache.

Figure 7.17 presents the stacked DRAM hit rate for the latency-optimized Alloy Cache [117], PoM and both Chameleon designs. Since Alloy Cache employs a

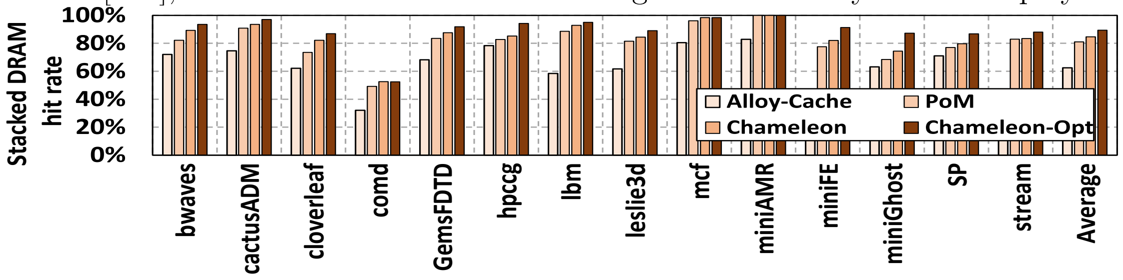


Figure 7.17: Stacked DRAM hit rate results.

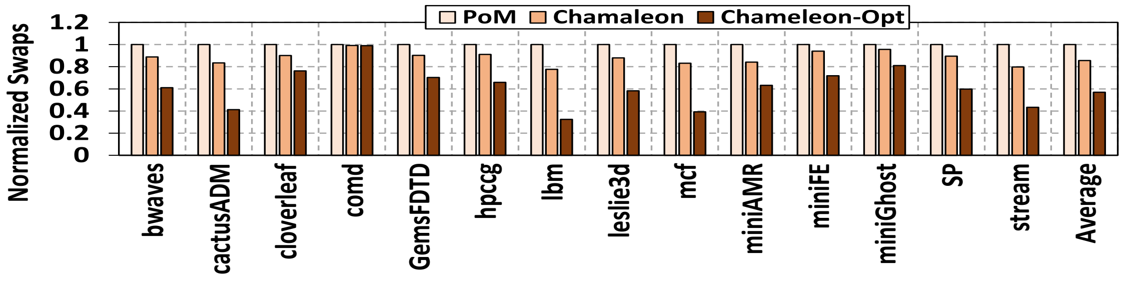


Figure 7.18: Normalized Swaps results.

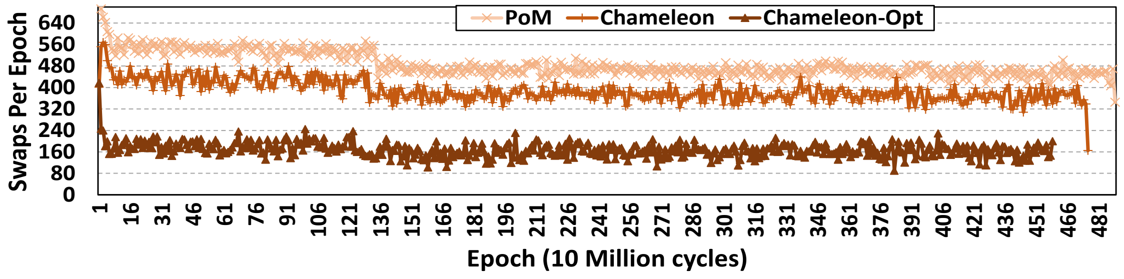


Figure 7.19: Swaps timeline for lbm workload.

latency-optimized direct-mapped cache design with 64B lines, it has the lowest stacked DRAM average hit rate of 62.4%, while PoM with 2KB segments has an average hit rate of 81%. In comparison, Chameleon and Chameleon-Opt have average hit rates of 84.6% of 89.4%, respectively. This higher hit rates in Chameleon and Chameleon-Opt can be attributed to more segment groups operating in cache mode. As discussed in Section 7.3.5, PoM employs a “threshold" which signifies the minimum number of accesses to an off-chip DRAM segment before it can be swapped with a stacked DRAM segment. Since Chameleon does not employ any such threshold for segment groups operating the in cache mode, it has a higher stacked DRAM hit rate compared to PoM. Since more such segment groups operate in cache mode in Chameleon-Opt, its hit rate is higher than Chameleon.

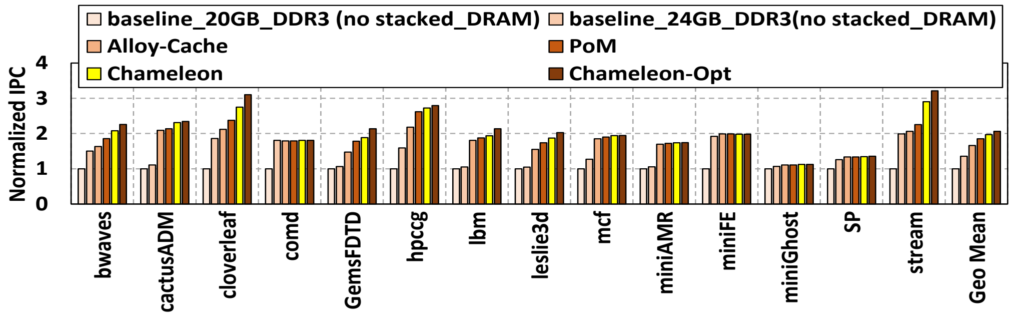


Figure 7.20: Normalized IPC results.

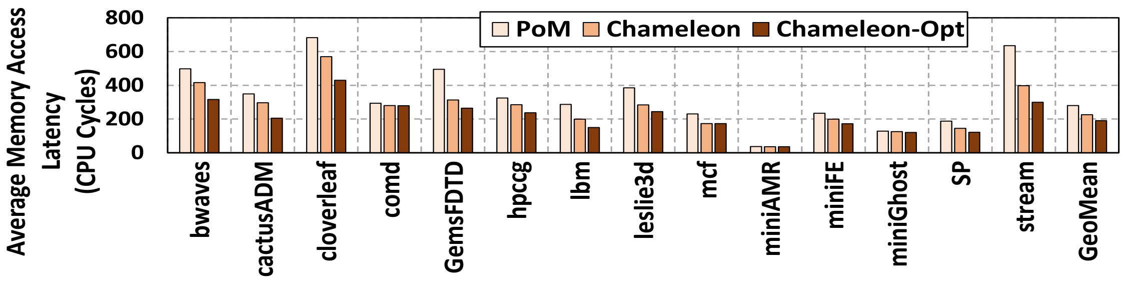


Figure 7.21: Average Memory Access Latency results.

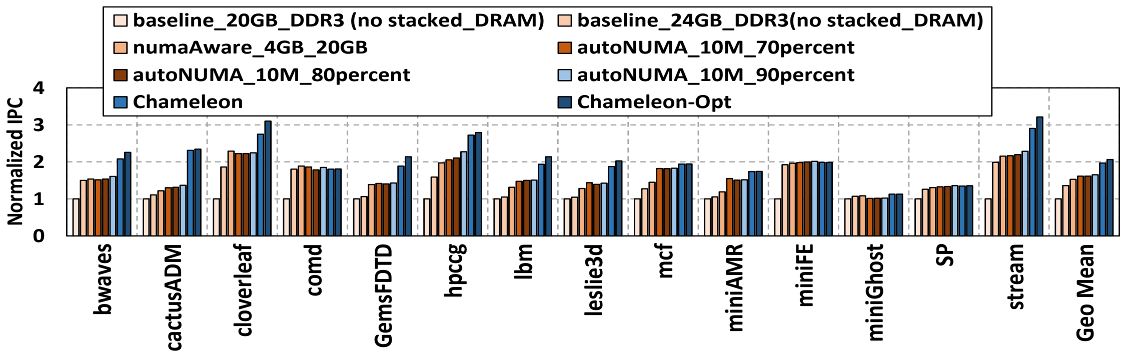


Figure 7.22: Normalized IPC results comparison.

Figure 7.18 quantifies the number of swaps incurred in PoM, Chameleon and Chameleon-Opt designs. The results reported are *normalized* to the number of swaps incurred in PoM. Due to many segment groups operating in the cache mode, the overall number of swaps is reduced on an average by 14.4% and 43.1% in Chameleon and Chameleon-Opt, respectively vs. PoM. Note that for a segment group operating in cache mode, evicting a modified (represented by dirty bit) stacked DRAM segment results in a writeback to off-chip DRAM before the stacked DRAM segment is filled with an off-chip segment. This is effectively still a swap, as the writeback of the modified segment still consumes both the stacked and off-chip memories' bandwidth. Hence in our Chameleon results reported, these scenarios are still considered as swaps. Figure 7.19 shows the number of swaps-per-epoch over time (each epoch 10 million CPU cycles) for lbm workload. Chameleon-Opt consistently experiences a lower number of swaps over PoM and Chameleon.

Figure 7.20 shows the normalized IPC of various designs including Alloy Cache, PoM, Chameleon, and Chameleon-Opt. There are two variant baseline systems, both without stacked DRAM and the total capacity coming from off-chip DRAM. While one baseline offers 20GB overall capacity, the other offers 24GB overall capacity. The 24GB baseline does not incur any page faults unlike the 20GB capacity. The 24GB baseline system improves the Geometric Mean of IPC by 35.6% over the 20GB capacity baseline. As mentioned in Section 7.6.1, Alloy Cache, PoM, Chameleon and Chameleon-Opt configurations all use a 4GB stacked and 20GB off-chip DRAM, thereby having a total capacity of 24GB. Alloy Cache experiences page faults for workloads with high memory footprints similar to both baselines since it sacrifices total capacity to use as a cache. As a result, though it improves the performance over the baselines, its performance is lower than other alternatives for many workloads, as can be observed in Figure 7.20. PoM improves the performance (Geometric Mean of IPC) over the 20GB and 24GB capacity baselines by 85.2% and 36.5%, respectively. Chameleon improves the Geometric Mean of IPC for all the workloads by 96.8% and 45.1% over the 20GB and 24GB baseline systems respectively, and by 6.3% and 18.5% over PoM and Alloy Cache, respectively. Chameleon-Opt improves the performance by 106.3% and 52.0% over the 20GB and 24GB baseline systems respectively, and by 11.6% and 24.2% over PoM and Alloy Cache, respectively. Such an improvement over the baseline systems is mainly because Chameleon manages to cater for the high memory footprint by averting page-faults, and could utilize the high bandwidth stacked DRAM more efficiently, averting slow off-chip DRAM accesses. The success of Chameleon and Chameleon-Opt over PoM is because of the increased stacked DRAM hit rate as well as the reduced number of swaps, which reduces the average memory access latency in Chameleon and Chameleon-Opt, as shown in Figure 7.21. Further, as Chameleon-Opt can retain/create more free space in the stacked DRAM compared to Chameleon, it achieves a higher hit rate with a lower number of swaps, thereby lowering the overall memory access latency. Hence, Chameleon-Opt improves the performance further by 4.8% over Chameleon.

7.6.3 Comparison with OS-based solutions

As discussed in Sections 7.3.1.1 and 7.3.1.2, the NUMA-Aware Memory Allocator and AutoNUMA under-utilize stacked DRAM resulting in an stacked DRAM hit rates of 18.5% and 64.4%. Therefore, these OS-based solutions do not leverage the full potential of high-bandwidth stacked DRAM. Figure 7.22 shows that Chameleon has an average improvement of 28.7% and 19.1% over NUMA-Aware Memory Allocator and AutoNUMA respectively, while Chameleon-Opt improves the performance

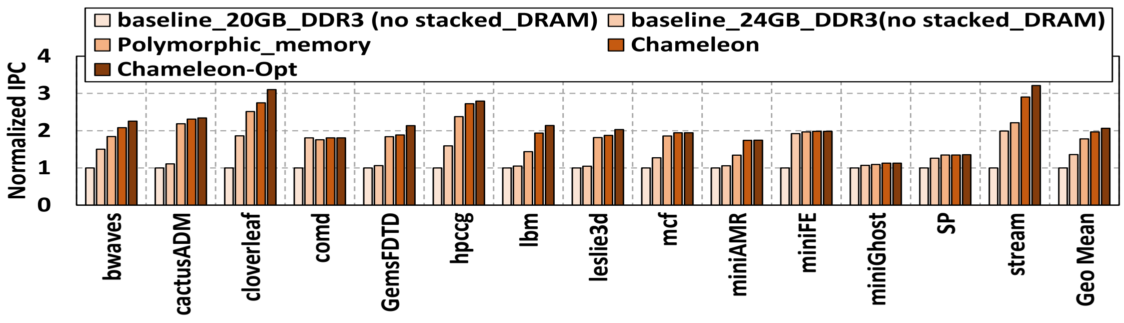


Figure 7.23: Polymorphic Memory [140] Comparison.

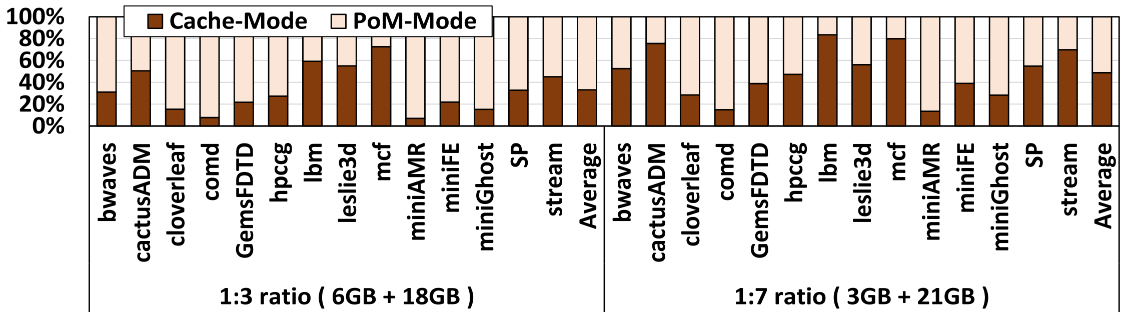


Figure 7.24: Sensitivity distribution results for Chameleon-Opt.

by 34.8% and 24.9%.

7.6.4 Comparison with Polymorphic Memory

Chung, et al. [140] proposed a hybrid architecture which can leverage the free memory available in stacked DRAM as cache. This patent is the closest work to our proposal. Their proposed architecture could leverage the OS-visible stacked DRAM free space as a cache, but it does not leverage the free space available in off-chip DRAM to be used as a cache (unlike Chameleon-Opt). Though this proposal achieves the same amount of free space as our original Chameleon design, our original Chameleon co-design still outperforms their proposal by 10.5% as can be observed in Figure 7.23. This is because Polymorphic Memory does not swap the most frequently used pages from the off-chip DRAM with the ones in stacked DRAM for OS allocated pages. Thus, apart from the cached segments, only pages allocated by OS in stacked DRAM incur stacked DRAM hits thereby under-utilizing the stacked DRAM. Chameleon and Chameleon-Opt improve the Geometric Mean of the IPC by 10.5% and 15.8% over Polymorphic Memory, respectively.

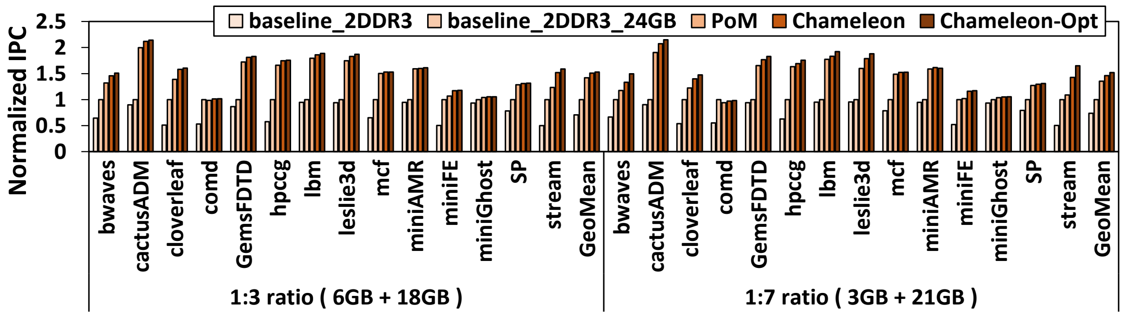


Figure 7.25: Sensitivity IPC results for Chameleon-Opt.

7.6.5 Sensitivity Results

Figure 7.24 shows the cache and PoM mode distribution for different ratios of stacked and off-chip DRAM capacities. For a 1:3 ratio, the stacked DRAM contributes a total capacity of 6GB, while the off-chip DRAM contributes 18GB and for 1:7 ratio, stacked DRAM contributes 3GB, while the off-chip contributes 21GB. As the ratio of stacked-to-off-chip DRAM increases from 1:3 to 1:7, the number of segment groups operating on an average increases from 33% to 48.7% vs 40.6% in 1:5 configuration. This is because as the number of segments per segment group increases from 3 to 7, the probability of finding at least one free segment increases in Chameleon-Opt thereby increasing the cache mode segment groups. Figure 7.25 represents the normalized performance corresponding to these ratios, and shows that Chameleon-Opt consistently performs better across all ratios. For 1:3 ratio, Chameleon and Chameleon-Opt improve the performance by 5.9% and 7.6% over PoM respectively, while in 1:7 ratio, Chameleon and Chameleon-Opt improve the performance by 8.1% and 12.4% over PoM, respectively.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

My thesis broadly looked at addressing the "memory wall" problem. At a high-level my dissertation can be classified in to a three-pronged approach: hardware-only, software-only and hardware-software co-design approaches to address the performance-gap between the processor and memory.

As a hardware approach, I proposed Re-NUCA, a complete hardware approach which wear-levels the Re-RAM based last-level caches in a performance-conscious manner in a manycore processor. Re-NUCA is based on the notion that not all the loads are critical for performance. As a result, Re-NUCA employs a hybrid NUCA mapping policy which wear-levels the cache banks such that the critical cache-lines are mapped using R-NUCA policy, while the non-critical cache lines are mapped using Static NUCA policy. By performing a performace-conscious wearleveling, Re-NUCA achieves best of both worlds interms of performance and lifetime.

Next in a large NUMA system, I proposed CAMM, a complete software approach evaluated in VMware ESXi which detects and alleviates congestion dynamically. CAMM estimates congestion by probing certain set of pages in a non-cacheable manner and based on the congestion detected, CAMM employs Congestion-Aware Memory Management (CAMA) or Congestion-Aware Page Migration (CAPM) or a mixture of both to alleviate congestion. Our CAMM does not warrant reading any performance counters and is successful in alleviating congestion quiet successfully with minimal overheads.

The scaling in DRAM chip density has undesirable overheads on the overall performance due to the inherent DRAM refresh overheads. To address these performance bottlenecks, I proposed a hardware-software co-design where hardware employed DRAM refresh schedule is changed slightly and exposed to the OS. Based on the updated refresh schedule, OS employs a soft-partitioned memory allocation strategy. The updated memory allocation in OS and the new DRAM refresh schedule in hardware further provided the avenues for OS to schedule a task that is minimally impacted by the DRAM refreshes, thereby alleviating the DRAM refresh overheads.

With the advent of emerging stacked DRAMs, the design decision of how to integrate these stacked DRAMs in to the systems became an interesting challenge for the computer architects. Prior designs have proposed using stacked DRAM as a hardware-managed cache or OS-visible extension to off-chip DRAM. Motivated by the fact that such static design decisions cannot unravel the full-potential of these stacked DRAMs, I proposed Chameleon, a dynamically reconfigurable memory systems. Based on the workloads currently executing on the system the Chameleon will dynamically reconfigure the heterogeneous memory system to operate certain sections of memory in the part of memory mode and other sections in the cache mode. The proposed Chameleon system showed good improvement in performance compared to a statically designed cache and part-of-memory systems.

8.2 Future Work

We are at an interesting juncture in the computer architecture era with plethora of emerging memory technologies available in the markets. Few such offerings include Intel 3D X-point, Micron's HMC, AMD's HBM memories to name a few. The integration of these memories at various levels in the un-core hierarchy to achieve maximum performance and energy-efficiency still has tremendous scope for improvement. As part of my future work, I will continue investigating in these directions especially at optimizing the overall system for energy-efficiency.

Appendix | Publications and Patents

1 Main Publications

[ISCA 2018, Under Submission]
Jagadish B.Kotra, Haibo Zhang, Alaa R. Alameldeen, Chris Wilkerson, Mahmut T. Kandemir
CHAMELEON: A Co-Design Based Dynamically Reconfigurable Heterogeneous Memory System,
In 45th ACM International Symposium on Computer Architecture (ISCA), 2018.

[ASPLOS 2017]
Jagadish B.Kotra, Narges Shahidi, Zeshan A. Chisthi, Mahmut T. Kandemir
Hardware-software co-design to mitigate DRAM refresh overheads. A case for refresh-aware process scheduling,
In 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2017.

[IISWC 2017]
Jagadish B.Kotra, Seongbeom Kim, Kamesh Madduri, Mahmut T. Kandemir
Congestion-Aware Memory Allocation and Migration Schemes for Virtualized NUMA Platforms: A VMware ESXi case study,
In IEEE International Symposium on Workload Characterization (IISWC), 2017.

[MASCOTS 2017]
Jagadish B.Kotra, Diana Guttman, Nachiappan Chidambaram, Mahmut T. Kandemir, Chita R. Das
Quantifying the Potential Benefits of Near-Data Computing in Manycores processors,
In 25th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), 2017.

[IPDPS 2016]
Jagadish B. Kotra, Mahommad Arjamond, Diana Guttman, Mahmut T. Kandemir, Chita R. Das
Re-NUCA: A Practical NUCA Architecture for ReRAM based last-level caches,
In International Parallel and Distributed Processing Symposium (IPDPS), 2016.

2 Other Significant Publications

[CVPR 2018, Under Submission]
Huaipan Jiang, Anup Sharma, Jihyun Ryoo, **Jagadish B. Kotra**, Meena Kandasamy, Chita R. Das, Mahmut Kandemir
A Tiling-based Hierarchial Neural Network Approach for Biomedical Image Segmentation,
CVPR, 2018.

[ISCA 2018, Under Submission]
Chun-yi Liu, **Jagadish B. Kotra**, Myoungsoo Jung, Mahmut Kandemir
CHIMERA: Designing a Hybrid Architecture for Reliable, High-Density 3D Flash Storage,
ISCA, 2018.

[ISCA 2018, Under Submission]
Sumitha George, Minli Liao, Huaipan Jiang, **Jagadish B. Kotra**, Mahmut

Kandemir,John Sampson, Vijaykrishnan Narayanan
MDACache: Caching for Multi-Dimensional-Access Memories,
ISCA, 2018.

[FAST 2018, Under Submission]
Chun-yi Liu, **Jagadish B. Kotra**, Myoungsoo Jung, Mahmut Kandemir
PEN: A Design of Partial-Erase for 3D NAND-based High Capacity SSDs,
FAST, 2018.

[PLDI 2018, Under Submission]
Orhan Kislal, **Jagadish B. Kotra**, Xulong Tang, Mahmut Kandemir, Mustafa Karakoy
Location-Aware Computation Mapping for Manycores,
PLDI, 2018.

[PACT 2017 (Poster)]
Orhan Kislal, **Jagadish B. Kotra**, Xulong Tang, Mahmut Kandemir, Mustafa Karakoy
Location-Aware Computation Mapping for Manycores,
In proceedings of The 26th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2017.

[Parlearning (IPDPS Workshop), 2016]
Orhan Kislal, Mahmut T. Kandemir, **Jagadish B. Kotra**
Cache-Aware Approximate Computing for Decision Tree Learning,
In International Workshop on Parallel and Distributed Computing for Large Scale Machine Learning and Big Data Analytics (ParLearning), 2016.

[MICRO 2016]
Xulong Tang, Mahmut T. Kandemir, Praveen Yedlapalli, **Jagadish B. Kotra**
Improving Bank-Level Parallelism for Irregular Applications,
In proceedings of The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2016.

[ICDCS 2015]
Joshua Dennis Booth, **Jagadish B. Kotra**, Hui Zhao, Mahmut T. Kandemir, Padhma Raghavan
Phase Detection with Hidden Markov Models for DVFS on Many-Core Processors,
In International Conference on Distributed Computing Systems (ICDCS), 2015.

[DAC 2015]
Jun Liu, **Jagadish B. Kotra**, Wei Ding, Mahmut T. Kandemir
Network Footprint Reduction through Data Access and Computation Placement in NoC-Based Manycores,
In Design Automation Conference (DAC), 2015.

[VLSI Design 2015]
Karthik Swaminathan, **Jagadish B. Kotra**, Hui Chu, Jack Sampson, Mahmut T. Kandemir, Vijay Narayanan
Thermal-Aware Application Scheduling on Device-Heterogeneous Embedded Architectures,
In International Conference on VLSI Design, 2015.

[PACT 2013]
Praveen Yedlapalli **Jagadish B. Kotra**, Emre Kultursay, Mahmut T. Kandemir, Anand Sivasubramaniam, Chita R. Das
Meeting midway: Improving CMP performance with memory-side prefetching,
In Parallel Architectures and Compilation Techniques (PACT), 2013.

3 Patents

[2016]
Jagadish B. Kotra, Alaa R. Alameldeen, Chris Wilkerson, Jaewoong Sim.
Multi-Level System Memory Having Near Memory Space Capable Of Behaving as Near Memory Cache or Fast Addressable System Memory Depending on System State,

Filed in USPTO (Intel).

[**2016**]
Jagadish B. Kotra, Zeshan A. Chisthi.
Hardware-software co-design for mitigating DRAM refresh overheads,
Filed in USPTO (Intel).

[**2015**]
Jagadish B. Kotra, Seongbeom Kim, Fei Guo.
Memory Congestion Aware NUMA Management,
Filed in USPTO (VMware).

Bibliography

[1] Sun, X.-H. (2006) "Remove the memory wall: from performance modeling to architecture optimization," in *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*.

[2] "IBM Power8 Processor," https://goo.gl/K9KsOO.

[3] "AMD HBM," `http://goo.gl/iVHexT`.

[4] Intel (2013), "The Intel Xeon Phi Product Family," .
URL `http://www.intel.com/content/www/us/en/high-performance-computing/high-performance-xeon-phi-coprocessor-brief.html`

[5] Chaudhuri, M. (2009) "PageNUCA: Selected policies for page-grain locality management in large shared chip-multiprocessor caches," in *HPCA*.

[6] Kim, C., D. Burger, and S. W. Keckler (2002) "An Adaptive, Non-uniform Cache Structure for Wire-delay Dominated On-chip Caches," in *ASPLOS*.

[7] Hardavellas, N. et al. (2009) "Reactive NUCA: Near-optimal Block Placement and Replication in Distributed Caches," ISCA.

[8] "Intel Westmere processor," `http://goo.gl/UGPBsI`.

[9] "Intel Haswell Processor," `http://goo.gl/xIiqKY`.

[10] "AMD Bulldozer," `http://goo.gl/otbg9J`.

[11] Rixner, S., W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens (2000) "Memory Access Scheduling," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA.

[12] Kaseridis, D., J. Stuecheli, and L. K. John (2011) "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.

[13] Yedlapalli, P., J. Kotra, E. Kultursay, M. Kandemir, C. R. Das, and A. Sivasubramaniam (2013) "Meeting midway: Improving CMP performance with memory-side prefetching," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT.

[14] Liu, J., B. Jaiyen, R. Veras, and O. Mutlu (2012) "RAIDR: Retention-Aware Intelligent DRAM Refresh," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA.

[15] Chang, K. K. W., D. Lee, Z. Chishti, A. R. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu (2014) "Improving DRAM performance by parallelizing refreshes with accesses," in *the 20th International Symposium on High Performance Computer Architecture*, HPCA.

[16] Mukundan, J., H. Hunter, K.-h. Kim, J. Stuecheli, and J. F. Martínez (2013) "Understanding and Mitigating Refresh Overheads in High-density DDR4 DRAM Systems," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA.

[17] Bhati, I., Z. Chishti, S.-L. Lu, and B. Jacob (2015) "Flexible Auto-refresh: Enabling Scalable and Energy-efficient DRAM Refresh Reductions," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA.

[18] "Understanding the Linux Kernel," http://goo.gl/8P7gJR.

[19] Liu, L., Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu (2012) "A Software Memory Partition Approach for Eliminating Bank-level Interference in Multicore Systems," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT.

[20] Yun, H., R. Mancuso, Z. P. Wu, and R. Pellizzoni (2014) "PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium*, RTAS.

[21] Jeong, M. K., D. H. Yoon, D. Sunwoo, M. Sullivan, I. Lee, and M. Erez (2012) "Balancing DRAM locality and parallelism in shared memory CMP systems," in *IEEE International Symposium on High-Performance Comp Architecture*, HPCA.

[22] "Linux CFS Scheduler," https://goo.gl/hjVjJl.

[23] Lozi, J., B. Lepers, J. R. Funston, F. Gaud, V. Quéma, and A. Fedorova (2016) "The Linux scheduler: a decade of wasted cores," in *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys.

[24] Fedorov, V. V., A. L. N. Reddy, and P. V. Gratz (2015) "Shared Last-Level Caches and The Case for Longer Timeslices," in *Proceedings of the 2015 International Symposium on Memory Systems*, MEMSYS.

[25] Kim, C. et al. (2003) "A forward body-biased-low-leakage SRAM cache: device and architecture considerations," in *ISLPED*.

[26] Sun, G. et al. (2009) "A novel architecture of the 3D stacked MRAM L2 cache for CMPs," in *HPCA*.

[27] Rasquinha, M. et al. (2010) "An energy efficient cache design using Spin Torque Transfer (STT) RAM," in *ISLPED*.

[28] Mishra, A. K. et al. (2011) "Architecting on-chip interconnects for stacked 3D STT-RAM caches in CMPs," in *ISCA*.

[29] Park, S. P. et al. (2012) "Future cache design using STT MRAMs for improved energy efficiency: Devices, circuits and architecture," in *DAC*.

[30] Sun, Z. et al. (2011) "Multi Retention Level STT-RAM Cache Designs with a Dynamic Refresh Scheme," in *MICRO*.

[31] Shevgoor, M. et al. (2015) "Improving Memristor Memory with Sneak Current Sharing," in *ICCD*.

[32] Xu, C. et al. (2015) "Overcoming the challenges of crossbar resistive memory architectures," in *HPCA*, pp. 476–488.

[33] Wei, Z. et al. (2008) "Highly reliable TaOx ReRAM and direct evidence of redox reaction mechanism," in *IEDM*.

[34] Kim, Y.-B. et al. (2011) "Bi-layered RRAM with unlimited endurance and extremely uniform switching," in *VLSIT*.

[35] Lee, M.-J. et al. (2011) "A fast, high-endurance and scalable non-volatile memory device made from asymmetric Ta2O5-x/TaO2-x bilayer structures," *Nature Materials*.

[36] Aratani, K. et al. (2007) "A Novel Resistance Memory with High Scalability and Nanosecond Switching," in *IEDM*.

[37] Spradling, C. D. (2007) "SPEC CPU2006 benchmark tools," *SIGARCH CAN*.

[38] Ghose, S. et al. (2013) "Improving Memory Scheduling via Processor-side Load Criticality Information," ISCA.

[39] Binkert, N. et al. (2011) "The Gem5 Simulator," .

[40] Mittal, S. and J. S. Vetter (2014) "EqualChance: Addressing Intra-set Write Variation to Increase Lifetime of Non-volatile Caches," INFLOW.

[41] Wang, J. et al. (2013) "i2WAP: Improving non-volatile cache lifetime by reducing inter- and intra-set write variations," in *HPCA*.

[42] "Westmere Config," `https://goo.gl/CjGOOX`.

[43] "NAS," `https://goo.gl/dA36i5`.

[44] "SPEC CPU 2006," `http://www.spec.org/cpu2006`.

[45] "SPEC OMP 2001," `https://www.spec.org/omp2001/`,.

[46] "HPCCG," `https://mantevo.org/default.php`.

[47] "SPECJBB," `https://www.spec.org/jbb2005/`.

[48] Slota, G. et al. (2014) "BFS and Coloring-Based Parallel Algorithms for Strongly Connected Components and Related Problems," in *IPDPS*.

[49] "VMware ESX Server 2," white paper.

[50] (2012), "JEDEC," DDR3 SDRAM Standard.

[51] (2012), "JEDEC," DDR4 SDRAM Standard.

[52] (2012), "JEDEC," Low Power Double Data Rate 3 (LPDDR3).

[53] "Intel MCDRAM," `https://goo.gl/iGQG32`.

[54] "Intel 3D XPoint Memory," `http://goo.gl/04n7Ksl`.

[55] "Phase-change Memory," `https://goo.gl/xOKMDF`.

[56] Blagodurov, S. et al. (2011) "A Case for NUMA-aware Contention Management on Multicore Systems," in *USENIX ATC*.

[57] Liu, M. et al. (2014) "Optimizing Virtual Machine Consolidation Performance on NUMA Server Architecture for Cloud Workloads," in *ISCA*.

[58] Srikanthan, S. et al. (2015) "Data Sharing or Resource Contention: Toward Performance Transparency on Multicore Systems," in *USENIX ATC*.

[59] Sharanyan et al. (2016) "Coherence Stalls or Latency Tolerance: Informed CPU Scheduling for Socket and Core Sharing," in *USENIX ATC*.

[60] "Intel Performance Analysis Guide," `https://goo.gl/y21oEB`,.

[61] Ferdman, M. et al. (2012) "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware," in *ASPLOS*.

[62] Wang, L. et al. (2014) "BigDataBench: A big data benchmark suite from internet services," in *HPCA*.

[63] Agarwal, N. et al. (2017) "Thermostat: Application-transparent Page Management for Two-tiered Main Memory," in *ASPLOS*.

[64] Gandhi, J. et al. (2014) "BadgerTrap: A Tool to Instrument x86-64 TLB Misses," in *CAN*.

[65] Villavieja, C. et al. (2011) "DiDi: Mitigating the Performance Impact of TLB Shootdowns Using a Shared TLB Directory," in *PACT*.

[66] "Linux AutoNUMA," `https://lwn.net/Articles/488709/`.

[67] Lucas, R. et al. (2014) "Top Ten Exascale Challenges," .

[68] Bennett, J., H. Abbasi, P. T. Bremer, R. Grout, A. Gyulassy, T. Jin, S. Klasky, H. Kolla, M. Parashar, V. Pascucci, P. Pebay, D. Thompson, H. Yu, F. Zhang, and J. Chen (2012) "Combining in-situ and in-transit processing to enable extreme-scale scientific analysis," in *2012 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[69] Docan, C. et al. (2011) "Moving the Code to the Data - Dynamic Code Deployment Using ActiveSpaces," in *IPDPS*.

[70] Zhang, F., M. Parashar, J. Cummings, and S. Klasky (2012) "Enabling In-situ Execution of Coupled Scientific Workflow on Multi-core Platform," in *2011 IEEE International Parallel Distributed Processing Symposium (IPDPS)*.

[71] Zheng, F., H. Abbasi, C. Docan, J. Lofstead, Q. Liu, S. Klasky, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf (2010) "PreDatA: preparatory data analytics on peta-scale machines," in *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*.

[72] "Intel MC-DRAM," https://goo.gl/Z4UiKo.

[73] Mattina, M., "The Architecture and Performance of the TILE-Gx Processor Family," .
URL `http://www.tilera.com/products/processors/TILE-Gx\_Family`

[74] Hackenberg, D., D. Molka, and W. E. Nagel (2009) "Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[75] Kestor, G., R. Gioiosa, D. J. Kerbyson, and A. Hoisie (2013) "Quantifying the energy cost of data movement in scientific applications," in *2013 IEEE International Symposium on Workload Characterization (IISWC)*.

[76] Hsieh, K., E. Ebrahim, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler (2016) "Transparent Offloading and Mapping (TOM): Enabling Programmer-transparent Near-data Processing in GPU Systems," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*.

[77] Pugsley, S., J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li (2014) "NDC: Analyzing the Impact of 3D-Stacked Memory+Logic Devices on MapReduce Workloads," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

[78] Farmahini-Farahani, A., J. H. Ahn, K. Morrow, and N. S. Kim (2015) "NDA: Near-DRAM Acceleration Architecture Leveraging Commodity DRAM Devices and Standard Memory Modules," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*.

[79] Kotra, J. B., N. Shahidi, Z. A. Chishti, and M. T. Kandemir (2017) "Hardware-Software Co-design to Mitigate DRAM Refresh Overheads: A Case for Refresh-Aware Process Scheduling," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[80] Rixner, S., W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens (2000) "Memory Access Scheduling," in *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*.

[81] Kumar, A., L.-S. Peh, P. Kundu, and N. K. Jha (2007) "Express Virtual Channels: Towards the Ideal Interconnection Fabric," in *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*.

[82] Venkata, S. K., I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor (2009) "SD-VBS: The San Diego Vision Benchmark Suite," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*.

[83] Bienia, C. (2011) *Benchmarking Modern Multiprocessors*, Ph.D. thesis, Princeton University.

[84] Stratton, J. A. et al. (2012), "IMPACT Technical Report," .
URL `http://impact.crhc.illinois.edu/Parboil/parboil.aspx`

[85] "The Scalable Heterogeneous Computing Benchmark Suite," .
URL `https://github.com/vetter/shoc-mic`

[86] Carlson, T. E. et al. (2011) "Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulations," in *Proc. of SC*.

[87] Stone, H. S. (1970) "A Logic-in-Memory Computer," *IEEE Trans. Comput.*

[88] Draper, J., , J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, and G. Daglikoca (2002) "The Architecture of the DIVA Processing-in-memory Chip," in *Proceedings of the 16th International Conference on Supercomputing (ICS)*.

[89] De, A., M. Gokhale, R. Gupta, and S. Swanson (2013) "Minerva: Accelerating Data Analysis in Next-Generation SSDs," in *Proceedings of the 2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*.

[90] Pugsley, S. H., J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li (2014) "Comparing Implementations of Near-Data Computing with In-Memory MapReduce Workloads." in *IEEE Micro*.

[91] Gao, M., J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis (2017) "TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[92] Chi, P., S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie (2016) "PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*.

[93] Qureshi, M. K., D. H. Kim, S. Khan, P. J. Nair, and O. Mutlu (2015) "AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems," in *IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN.

[94] Liu, S., K. Pattabiraman, T. Moscibroda, and B. G. Zorn (2011) "Flikker: Saving DRAM Refresh-power Through Critical Data Partitioning," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS.

[95] Venkatesan, R. K., S. Herr, and E. Rotenberg (2006) "Retention-aware placement in DRAM (RAPID): software methods for quasi-non-volatile DRAM," in *The Twelfth International Symposium on High-Performance Computer Architecture*, HPCA.

[96] "STREAM," https://www.cs.virginia.edu/stream/.

[97] "NAS," https://www.nas.nasa.gov/publications/npb.html.

[98] "Linux debugfs," https://goo.gl/sdBhIh.

[99] "Linux cgroups," http://goo.gl/tTiwSl.

[100] Chatterjee, N. et al. (2012) "Staged Reads: Mitigating the Impact of DRAM Writes on DRAM Reads," in *HPCA*.

[101] Binkert, N., B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood (2011) "The Gem5 Simulator," *SIGARCH Comput. Archit. News.*

[102] Poremba, M. and Y. Xie (2012) "NVMain: An Architectural-Level Main Memory Simulator for Emerging Non-volatile Memories," in *IEEE Computer Society Annual Symposium on VLSI*, ISVLSI.

[103] Stuecheli, J., D. Kaseridis, H. C. Hunter, and L. K. John (2010) "Elastic Refresh: Techniques to Mitigate Refresh Penalties in High Density Memory," in *the 43rd Annual International Symposium on Microarchitecture*, MICRO.

[104] Bhati, I., Z. Chishti, and B. Jacob (2013) "Coordinated Refresh: Energy Efficient Techniques for DRAM Refresh Scheduling," in *Proceedings of the 2013 International Symposium on Low Power Electronics and Design*, ISLPED.

[105] Nair, P., C. C. Chou, and M. K. Qureshi (2013) "A case for Refresh Pausing in DRAM memory systems," in *IEEE 19th International Symposium on High Performance Computer Architecture*, HPCA.

[106] Zhang, T., M. Poremba, C. Xu, G. Sun, and Y. Xie (2014) "CREAM: A Concurrent-Refresh-Aware DRAM Memory architecture," in *the 20th International Symposium on High Performance Computer Architecture*, HPCA.

[107] "Micron HMC," https://goo.gl/F3kXvm.

[108] Macri, J. (2015) "AMD's next generation GPU and high bandwidth memory architecture: FURY," in *2015 IEEE Hot Chips 27 Symposium (HCS)*.

[109] "AMD Vega," https://goo.gl/moSqgH.

[110] Chou, C., A. Jaleel, and M. K. Qureshi (2015) "BEAR: Techniques for Mitigating Bandwidth Bloat in Gigascale DRAM Caches," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*.

[111] Jang, H., Y. Lee, J. Kim, Y. Kim, J. Kim, J. Jeong, and J. W. Lee (2016) "Efficient footprint caching for Tagless DRAM Caches," in *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[112] Jevdjic, D., G. H. Loh, C. Kaynak, and B. Falsafi (2014) "Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[113] Jevdjic, D., S. Volos, and B. Falsafi (2013) "Die-stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*.

[114] Jiang, X., N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian (2010) "CHOP: Adaptive filter-based DRAM caching for CMP server platforms," in *In Proceedings of the 16th International Symposium on High-Performance Computer Architecture (HPCA)*.

[115] Lee, Y., J. Kim, H. Jang, H. Yang, J. Kim, J. Jeong, and J. W. Lee (2015) "A Fully Associative, Tagless DRAM Cache," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*.

[116] Loh, G. H. and M. D. Hill (2011) "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)*.

[117] Qureshi, M. K. and G. H. Loh (2012) "Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[118] Shin, S., S. Kim, and Y. Solihin (2016) "Dense Footprint Cache: Capacity-Efficient Die-Stacked DRAM Last Level Cache," in *Proceedings of the Second International Symposium on Memory Systems (MEMSYS)*.

[119] Sim, J., G. H. Loh, H. Kim, M. O'Connor, and M. Thottethodi (2012) "A Mostly-Clean DRAM Cache for Effective Hit Speculation and Self-Balancing Dispatch," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[120] Chou, C., A. Jaleel, and M. K. Qureshi (2014) "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)*.

[121] Oskin, M. and G. H. Loh (2015) "A Software-Managed Approach to Die-Stacked DRAM," in *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*.

[122] Ryoo, J. H., K. Ganesan, Y. M. Chen, and L. K. John (2015) "i-MIRROR: A Software Managed Die-Stacked DRAM-Based Memory Subsystem," in *Proceedings of the 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*.

[123] Sim, J., A. R. Alameldeen, Z. Chishti, C. Wilkerson, and H. Kim (2014) "Transparent Hardware Management of Stacked DRAM As Part of Memory," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[124] "ESXi White paper," https://goo.gl/5tQQBk.

[125] "NUMA-aware Allocation," https://goo.gl/PrRgLQ.

[126] Gulur, N., M. Mehendale, R. Manikantan, and R. Govindarajan (2014) "Bi-Modal DRAM Cache: Improving Hit Rate, Hit Latency and Bandwidth," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*.

[127] Meswani, M. R., S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh (2015) "Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*.

[128] Yu, X., C. J. Hughes, N. Satish, O. Mutlu, and S. Devadas (2017) "Banshee: Bandwidth-Efficient DRAM Caching Via Software/Hardware Cooperation," *CoRR*, **abs/1704.02677**.

[129] Prodromou, A., M. Meswani, N. Jayasena, G. Loh, and D. M. Tullsen (2017) "MemPod: A Clustered Architecture for Efficient and Scalable Migration in Flat Address Space Multi-level Memories," in *Proceedings of the 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[130] "Linux numactl," `https://goo.gl/FS9mHs`.

[131] "Intel Xeon Machine," https://goo.gl/USH8dD.

[132] "Rate Mode," https://goo.gl/i4148Z.

[133] "NAS Benchmark," https://goo.gl/jQvMKbl.

[134] "numastat," https://goo.gl/M8U5hN.

[135] "Traditional Pages," https://goo.gl/zgCHKQ.

[136] "Transparent Huge Pages," https://goo.gl/nrqpRT.

[137] "Giant Pages," https://goo.gl/BE6FtN.

[138] "Linux Buddy Allocator," https://goo.gl/bvpcuj.

[139] "GEM5 Pseudo-instructions," https://goo.gl/XDzQcw.

[140] Chung, J. and N. Soundararajan (2012), "Polymorphic Stacked DRAM Memory Architecture," US Patent App. 13/036,839.
URL `https://www.google.ch/patents/US20120221785`

Vita

Jagadish B. Kotra

Jagadish B. Kotra is a Ph.D student in the Department of Computer Science and Engineering at Pennsylvania State University. Before starting his Ph.D at Penn State, Jagadish worked at IBM Software Labs in Bangalore, India, working on IBM Middleware Products and IBM's [J9]JVM. Even before his stint at IBM, Jagadish obtained his B.Tech degree from Acharya Nagarjuna University in India. Jagadish's research interests lie in the broad area of Computer Architecture, Operating Systems and Hardware-Software co-design. Specifically, Jagadish's PhD thesis aimed at bridging the performance-gap between Processor and Memory using hardware-only, software-only and hardware-software co-design techniques. His thesis looked at enabling seamless integration of emerging memory technologies like Resistive RAMs (Re-RAMs) and 3D stacked DRAMs in to the memory hierarchy. As a Ph.D student at Penn State, Jagadish published extensively in conferences including: ASPLOS, IISWC, IPDPS, MASCOTS among others. As a graduate student, Jagadish interned in different places include Intel Labs, VMware and Intel product divisions. During his free-time, Jagadish enjoys playing raquet sports including Raquet-ball, Squash, Badminton and Tennis.