Paving the Path for Heterogeneous Memory Adoption in Production Systems

by

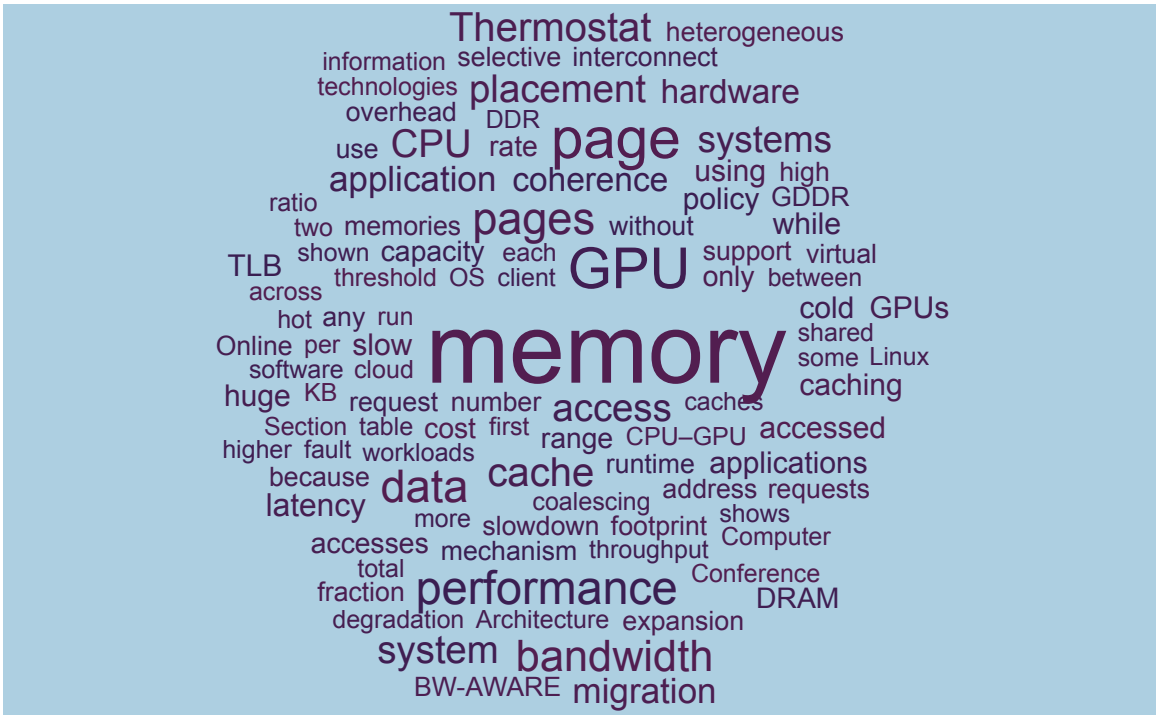
Neha Agarwal

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2017

Doctoral Committee:

Associate Professor Thomas F. Wenisch, Chair
Professor Ella M. Atkins
Professor Peter M. Chen
Assistant Professor Ronald G. Dreslinski Jr.

Neha Agarwal
nehaag@umich.edu
ORCID iD: 0000-0001-9833-6538


Thermostat
heterogeneous
information
selective
interconnect
technologies
placement
hardware
overhead
DDR
use
CPU
rate
page
systems
application
coherence
using
high
ratio
policy
GDDR
two
memories
pages
without
while
TLB
shown
capacity
each
GPU
support
virtual
across
threshold
OS
client
only
between
hot
any
run
cold
GPUs
Online
per
slow
memory
shared
software
cloud
some
Linux
huge
KB
request
number
access
caches
caching
Section
table
cost
first
range
CPU–GPU
accessed
higher
fault
workloads
because
data
cache
runtime
applications
latency
coalescing
address
requests
more
slowdown
footprint
shows
accesses
mechanism
throughput
Computer
total
Conference
fraction
performance
DRAM
degradation
Architecture
expansion
system
bandwidth
BW-AWARE
migration

To – Family

ACKNOWLEDGEMENTS

First and foremost I want to sincerely thank Tom, my PhD advisor, who gave me the opportunity to pursue the doctoral program with his guidance. Tom is a fantastic advisor, a statement that I have been saying time and again especially new students seeking an advisor. In Indian culture we have a special place and respect for the “guru” and I feel that I was fortunate enough to find Tom and to be able to associate with him as my “guru”. Tom is a highly energetic, optimistic, professional, and extremely knowledgeable person. His advising style is very adaptive and he let’s each student learn and work at his/her own pace. He does not create work pressure, but if you are going off tracks he will for sure warn you. He is an extremely adaptive and flexible advisor be it technically, administratively, or logistically. Because of his support I was able to explore different positions in industry and understand the contrasting but complementary research approaches in academia versus industry. With his guidance I have learnt to deal with rejections and failures with a positive attitude, which is an instrumental medicine for PhD journey. No matter how many rejections you get, his philosophy of keep trying to improve and doing excellent quality research is what I found most useful as his student. I am thankful to University of Michigan, CSE department to give me a wonderful opportunity of associating with Tom, as my advisor.

I also want to take this chance to thank all of my collaborators for their contributions in making my ideas more concrete and provide me the perspective that I lacked before. I especially want to mention about the guidance and support I got from David Nellans. Dave was my internship mentor at NVIDIA. His approach of conducting product influential research helped me understand impact of research on products and business utility for a company. Dave is also a very good friend who boosts my moral and motivates me to keep doing good work. Andrés Lagar-Cavilla is another important name in my technical career, because of whom I got interested in Linux and operating systems. His commitment to help and support his peers inspires me. His help during my internship at Google was the stepping stone for my success in my internship project.

I think coming into the PhD program was one of the best decisions of my life. I got many opportunities to travel in the last six years, exploring different countries, cultures,

and lifestyles. Travel gave me a perspective to meaning of life. I could go out of my comfort zone, experience people's generosity, overcome difficulty of not knowing whereabouts of a place. I realized that world is a much bigger and beautiful place and that experience matters more than success. Travel inspired me to explore new avenues in my research, become comfortable at switching projects and keep learning new things.

My parents – Madhu Agarwal and Bal Krishna Agarwal – are the reason whatever I am today. Their constant guidance and support in life has been instrumental in my writing this acknowledgement section today. They have provided me the shade and comfort at each and every single step in my life even when I didn't ask for it. They always understand what I need and are always there for me. The feeling of having them no matter what happened in my professional or personal life is the biggest gift I have gotten in my life. When things were not going the way I wanted them to go for very long time and I was doubting myself all the time, mummy papa were the one who always filled me with confidence and encouraged me to try even harder. I have called them several times late at nights to talk about how worried I am, how things are not happening even though I am doing all I can, they are always there to listen to me quietly and help me move forward. I can't imagine doing anything without their support. Thanks is a small word for what my parents have done for me. I can just say that I am lucky enough to have them in my life. Mummy–Papa this dissertation is your hard work and you confidence in me to be able to sincerely work and contribute to the world.

Next, I want to talk about my brother, Shyam Mohan Agarwal. He is the person who loves me so much that at times I think what did I do to receive that much love. He is the one because of whom I am able to fulfill my dream of studying in the best institutes of knowledge. His encouragement and belief in me is why I have the courage to try out new things fearlessly. He looked something in me since childhood and he knew that I would be a scholar. It has been like he has foreseen what I will become and then gave me a push to go and get it. With every achievement of mine it feels like he actually won something. He is the one who enjoys my success the most. Without him experiencing the joy of what I do my work is incomplete. He is a true inspiration to me, without whom I am not sure how I would have a direction to pursue something that I love most doing. I also want to thank my sister-in-law, Khyati Agarwal, who has been so nice to accommodate all of us in her life and continued to make our family loving and happy.

I have also been very fortunate to have come across several fantastic people and be friends with them. Gaurav, Pooja, Kataria, Gauri, Tanvi, Mohit, Kunal, Pulkit are some names very close to my heart that I know will come for me if I need them at any point in life. They have challenged me in several ways, making me a better person with every

interaction. Countless nights we have just sat together and just discussed some weird mystery of life or just cribbed about how things are unfair. These people are treasures I have found in my life and have made my life so comfortable and loving. They have played a significant role by keeping me sane during the most self-doubting periods of PhD journey. All of my seniors Gaurav Chadha, Ankit Sethia, Daya Khudia, and Abhayendra Singh are some of the people who helped me acclimatize in Ann Arbor and the department during my foundational years of my PhD.

I want to thank my labmates for being very supportive all throughout my graduate school. Steve Pelley has been a mentor to me and I always go to him to ask for all sorts of career advice. Faissal Sleiman is a hard working person who always inspires me to do the right thing. Akshitha, Vaibhav, Amlan, Aasheesh, Jeff all have been some of the fantastic, smart, and hard working individuals I have come across, who are like extended family to me. I also want to acknowledge CSE staff members Dawn Freysinger, Lauri Johnson, Karen Liska, Denise Duprie, and Stephen Reger who have helped in every possible way whenever I needed administrative/logistics support from the department.

Finally, I would like to mention the most significant person Subhasis Das, because of whom I have been able to survive the graduate school. He is the most amazing person that has happened to me in this life after having blessed with a fantastic family. This is the person without whom last five years would have been extremely hard. Even after living miles away he can bring smile on my face in seconds and make everything look good again. His presence in my life is priceless and I can't thank God enough for bringing him in my life. I will cherish all the fun we had discussing technical stuff, watching Netflix together and a getaway from cold Ann Arbor. I have learnt a lot from Subhasis's technical approach and his philosophy that at times you got to try million things before something actually starts working or is meaningful. Thanks so much for being part of my life, without you this journey will be incomplete.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGEMENTS iii

LIST OF FIGURES ix

LIST OF TABLES xii

ABSTRACT xiii

CHAPTER

I. Introduction 1

1.1 Bandwidth-asymmetric Systems 3
1.2 Different Coherence Domains 5
1.3 Cheaper Memory Technologies 6
1.4 Contributions 7
1.5 Impact 8

II. Background and Motivation 10

2.1 Bandwidth Hungry Characteristics of GPUs 10
2.2 Current OS NUMA Page Placement 12
2.3 Memory Copy Overhead in GPUs 14
2.4 Cache Coherent GPUs Enhancing System Programmability 15
2.5 Supporting Hardware Cache Coherence in GPUs 16
2.6 Virtual Memory Management in Linux 19
2.7 Transparent Huge Pages 19

III. Page Placement Strategies for GPUs within Heterogeneous Memory Systems 21

3.1 Introduction 21
3.2 BW-AWARE Page Placement 22

3.2.1 Bandwidth Maximized Page Placement . . . 23
3.2.2 Methodology . . . 25
3.2.3 BW-AWARE Performance: experimental results . . . 27
3.3 Understanding Application Memory Use . . . 30
3.3.1 Visualizing Page Access Distribution . . . 31
3.3.2 Oracle Page Placement . . . 35
3.4 Application Aware Page Placement . . . 36
3.4.1 Profiling Data Structure Accesses in GPU Programs . . . 37
3.4.2 Memory Placement APIs for GPUs . . . 38
3.4.3 Program Annotation For Data Placement . . . 38
3.4.4 Experimental Results . . . 40
3.4.5 Discussion . . . 41
3.5 Conclusion . . . 43

IV. Unlocking Bandwidth for GPUs in CC-NUMA Systems . . . 44

4.1 Introduction . . . 44
4.2 Balancing Page Migration and Cache-Coherent Access . . . 44
4.2.1 Methodology . . . 46
4.2.2 Results . . . 47
4.3 Range Expanding Migration Candidates . . . 50
4.3.1 Avoiding TLB Shootdowns With Range Expansion . . . 52
4.3.2 Results . . . 53
4.4 Bandwidth Balancing . . . 55
4.4.1 Results . . . 58
4.5 Conclusion . . . 60

V. Selective GPU Caches to Eliminate CPU–GPU Cache Coherence . . . 62

5.1 Introduction . . . 62
5.2 GPU Selective Caching . . . 64
5.2.1 Naive Selective Caching . . . 65
5.2.2 Improving Selective Caching Performance . . . 66
5.2.3 Promiscuous Read-Only Caching . . . 70
5.3 Methodology . . . 72
5.4 Results . . . 74
5.4.1 Microarchitectural Enhancements . . . 74
5.4.2 Promiscuous GPU Caching . . . 79
5.4.3 Discussion . . . 80
5.5 Conclusion . . . 82

VI. Thermostat: Application-transparent Page Management for Two-tiered Main Memory . . . 83

6.1 Introduction . . . 83

6.2 Motivation . . . 86
6.2.1 Shifting cold data to cheap memory . . . 86
6.2.2 Benefits of transparent huge pages . . . 88
6.3 Thermostat . . . 89
6.3.1 Overview . . . 89
6.3.2 Page sampling . . . 90
6.3.3 Page access counting . . . 90
6.3.4 Page classification . . . 91
6.3.5 Correction of mis-classified pages . . . 92
6.3.6 Migration of cold pages to slow memory . . . 93
6.3.7 Thermostat example . . . 93
6.4 Methodology . . . 94
6.4.1 System Configuration . . . 94
6.4.2 Emulating slow memory: BadgerTrap . . . 95
6.4.3 Benchmarks . . . 95
6.4.4 Runtime overhead of Thermostat Sampling . . . 97
6.5 Evaluation . . . 97
6.5.1 Sensitivity to tolerable slowdown target . . . 102
6.5.2 Migration overhead and slow memory access rate . . . 104
6.5.3 DRAM Cost Analysis . . . 105
6.6 Discussion . . . 105
6.6.1 Hardware support for access counting . . . 105
6.7 Conclusion . . . 107

VII. Related Work . . . 108

7.1 Page Placement And Migration . . . 108
7.2 Cache Coherence . . . 109
7.3 Two-level Memory . . . 111

VIII. Conclusion . . . 114

BIBLIOGRAPHY . . . 117

LIST OF FIGURES

Figure

1.1 System architectures for legacy, current, and future mixed GPU-CPU systems. 3
2.1 GPU performance sensitivity to bandwidth and latency changes. 11
2.2 GPU performance sensitivity to memory subsystem performance where GDDR provides 200GB/s, DDR provides 80GB/s, and `memcpy` bandwidth is 80GB/s. 14
2.3 GPU performance sensitivity to L1 and L2 latency and bandwidth changes. 16
2.4 Linux page table structure for X86 both with and without a transparent huge page. 18
3.1 BW-Ratio of high-bandwidth vs high-capacity memories for likely future HPC, desktop, and mobile systems 22
3.2 GPU workload performance with different page placement policies. xC-yB policy represents $x : y$ data transfer ratio from CO and BO memory respectively. 26
3.3 Performance of BW-AWARE placement as application footprint exceeds available high-bandwidth memory capacity. 28
3.4 Performance comparison between BW-AWARE, INTERLEAVE, and LOCAL page placement policies while varying the memory bandwidth ratio. . 29
3.5 Data bandwidth cumulative distribution function with pages sorted from hot (most memory accesses) to cold (least memory accesses). 31
3.6 `bfs`: CDF of data footprint versus virtual address data layout. 32
3.7 `mummergpu`: CDF of data footprint versus virtual address data layout. . . . 33
3.8 `needle`: CDF of data footprint versus virtual address data layout. 34
3.9 Oracle application performance of a constrained capacity system vs unconstrained capacity system 35
3.10 Annotated pseudo-code to do page placement at runtime taking into account relative hotness of data structures and data structure sizes 39
3.11 Profile-driven annotated page placement performance relative to INTERLEAVE, BW-AWARE and oracular policies under at 10% capacity constraint. 40
3.12 Annotated page placement effectiveness versus data sets variation after training phase under at 10% capacity constraint. 42
4.1 Opportunity cost of relying on cache coherence versus migrating pages near beginning of application run. 45

4.2 Performance of applications across varying migration thresholds, where threshold-N is the number of touches a given page must receive before being migrated from CPU-local to GPU-local memory. . . . 48
4.3 Performance overhead of GPU execution stall due to TLB shootdowns when using a first touch migration policy (threshold-1). . . . 50
4.4 Cumulative distribution of memory bandwidth versus application footprint. Secondary axis shows virtual page address of pages touched when sorted by access frequency. (`xsbench`) . . . 52
4.5 Effect of range expansion on workload performance when used in conjunction with threshold based migration. . . . 53
4.6 `bfs`: Fraction of total bandwidth serviced by GDDR during application runtime when when using thresholding alone (TH), then adding range expansion (TH+RE) and bandwidth aware migration (TH+RE+BWB). . . . 56
4.7 `xsbench`: Fraction of total bandwidth serviced by GDDR during application runtime when when using thresholding alone (TH), then adding range expansion (TH+RE) and bandwidth aware migration (TH+RE+BWB). . . 57
4.8 `needle`: Fraction of total bandwidth serviced by GDDR during application runtime when when using thresholding alone (TH), then adding range expansion (TH+RE) and bandwidth aware migration (TH+RE+BWB). . . 58
4.9 Application performance when using thresholding alone (TH), thresholding with range expansion (TH+RE), and thresholding combined with range expansion and bandwidth aware migration (TH+RE+BWB). . . . 59
4.10 Distribution of memory bandwidth into demand data bandwidth and migration bandwidth . . . 60
5.1 Number of coherent caches in future two socket CPU-only vs CPU–GPU systems. . . . 63
5.2 Overview of naive selective caching implementation and optional performance enhancements. Selective caching GPUs maintain memory coherence with the CPU while not requiring hardware cache coherence within the GPU domain. . . . 64
5.3 Fraction of 4KB OS pages that are read-only and read-write during GPU kernel execution. . . . 71
5.4 GPU performance under selective caching with uncoalesced requests, L1 coalesced requests, L1+L2 coalesced requests. . . . 75
5.5 GPU performance with selective caching when combining request coalescing with on CPU-side caching for GPU clients at 64KB–1MB cache capacities. (CC: Client-Cache) . . . 76
5.6 GPU data transferred across CPU-GPU interconnect (shown left y-axis) and performance (shown right y-axis) for 128B cache line-size link transfers and variable-size link transfers respectively. . . . 77
5.7 GPU performance when using Selective caching (Request-Coalescing + 512kB-CC + Variable-Transfers) combined with read-only based caching. geomean*: Geometric mean excluding `backprop, cns, needle`, where read-only caching would be switched-off. (RO: Read-Only) . . . 78

5.8 GPU performance under memory capacity constraints. (CC: Client-Cache, VT: Variable-sized Transfer Units) . . . 81
6.1 Fraction of 2MB pages idle/cold for 10s detected via per-page *Accessed* bits in hardware. Note that this technique cannot estimate the page access rate, and thus cannot estimate the performance degradation caused by placing these pages in slow memory (which exceeds 10% for Redis). . . . 84
6.2 Memory access rate vs. hardware *Accessed* bit distribution of 4KB regions within 2MB pages for Redis. The single hardware *Accessed* bit per page does not correlate with memory access rate per page, and thus cannot estimate page access rates with low overhead. . . . 87
6.3 Slow memory access rate over time. For a 3% tolerable slowdown and 1us slow memory access latency, the target slow memory access rate is 30K accesses/sec. Thermostat tracks this 30K accesses/sec target. Note that different benchmarks have different run times; we plot the x-axis for 1200 seconds. . . . 92
6.4 Thermostat operation: Thermostat classifies huge pages into hot and cold by randomly splitting a fraction of pages and estimating huge page access rate by poisoning a fraction of 4K pages within a huge page. The sampling policy is described in Section 6.3.2, Section 6.3.3 describes our approach to page access counting, and Section 6.3.4 describes the page classification policy. Note that the sampling and poisoning fractions here are for illustration purposes only. In our evaluation we sample 5% of huge pages and poison at most 50 4KB pages from a sampled huge page. . . . 93
6.5 Amount of cold data in Cassandra identified at run time with 2% throughput degradation for a write-heavy workload (5:95 read/write ratio). . . . 98
6.6 Amount of cold data in MySQL-TPCC identified at run time with 1.3% throughput degradation. . . . 99
6.7 Amount of cold data in Aerospike identified at run time with 1% throughput degradation for a read-heavy workload (95:5 read/write ratio). . . . 100
6.8 Amount of cold data in Redis identified at run time with 2% throughput degradation. . . . 101
6.9 Amount of cold data in in-memory analytics benchmark identified at run time with 3% runtime overhead. . . . 102
6.10 Amount of cold data in web-search benchmark identified at run time with 1% throughput and no 99th percentile latency degradation. . . . 103
6.11 Amount of cold data in identified at run time varying with the specified tolerable slowdown. All the benchmarks meet their performance targets (not shown in the figure) while placing cold data in the slow memory. . . . 104

LIST OF TABLES

Table

2.1 GPU L1 and L2 cache hit rates (average). 17
3.1 System configuration for heterogeneous CPU-GPU system. 24
4.1 Effectiveness of range prefetching at avoiding TLB shootdowns and runtime savings under a 100-cycle TLB shootdown overhead. 54
5.1 Percentage of memory accesses that can be coalesced into existing in-flight memory requests, when using L1 (intra-SM) coalescing, and L1 + L2 (inter-SM) coalescing. 68
5.2 Utilization of 128B cache line requests where the returned data must be discarded if there is no matching coalesced request. 69
5.3 Parameters for experimental GPGPU based simulation environment. 73
6.1 Throughput gain from 2MB huge pages under virtualization relative to 4KB pages on both host and guest. 86
6.2 Application memory footprints: resident set size and file-mapped pages. . . 97
6.3 Data migration rate and false classification rate of slow memory. These rates are well-below expected bandwidth of near-future memory technologies. . . 104
6.4 Memory spending savings relative to an all-DRAM system when using slow memory with different cost points relative to DRAM. 105

ABSTRACT

Paving the Path for Heterogeneous Memory Adoption in Production Systems
by
Neha Agarwal

Chair: Thomas F. Wenisch

Systems from smartphones to data-centers to supercomputers are increasingly heterogeneous, comprising various memory technologies and core types. Heterogeneous memory systems provide an opportunity to suitably match varying memory access patterns in applications, reducing CPU time thus increasing performance per dollar resulting in aggregate savings of millions of dollars in large-scale systems. However, with increased provisioning of main memory capacity per machine and differences in memory characteristics (for example, bandwidth, latency, cost, and density), memory management in such heterogeneous memory systems poses multi-fold challenges on system programmability and design.

In this thesis, we tackle memory management of two heterogeneous memory systems: (a) CPU-GPU systems with a unified virtual address space, and (b) Cloud computing platforms that can deploy cheaper but slower memory technologies alongside DRAMs to reduce cost of memory in data-centers. First, we show that operating systems do not have sufficient information to optimally manage pages in bandwidth-asymmetric systems and thus fail to maximize bandwidth to massively-threaded GPU applications sacrificing GPU throughput. We present BW-AWARE placement/migration policies to support OS to make optimal data management decisions. Second, we present a CPU-GPU cache coherence design where CPU and GPU need not implement same cache coherence protocol but provide cache-coherent memory interface to the programmer. Our proposal is first practical approach to provide a unified, coherent CPU–GPU address space without requiring hardware cache coherence, with a potential to enable an explosion in algorithms that leverage tightly coupled CPU–GPU coordination.

Finally, to reduce the cost of memory in cloud platforms where the trend has been to map datasets in memory, we make a case for a two-tiered memory system where cheaper (per bit) memories, such as Intel/Microns 3D XPoint, will be deployed alongside DRAM. We present Thermostat, an application-transparent huge-page-aware software mechanism to place pages in a dual-technology hybrid memory system while achieving both the cost advantages of two-tiered memory and performance advantages of transparent huge pages. With Thermostat's capability to control the application slowdown on a per application basis, cloud providers can realize cost savings from upcoming cheaper memory technologies by shifting infrequently accessed cold data to slow memory, while satisfying throughput demand of the customers.

CHAPTER I

Introduction

Heterogeneity is ubiquitous in systems ranging from supercomputers and data-centers to smartphones. The fastest supercomputers in the world incorporate accelerators such as graphics processing units (GPUs) and Xeon Phi co-processors alongside central processing units (CPUs) [1]. Similarly, data-centers deploy GPUs and field-programmable gate arrays (FPGA) for accelerating applications like web search and machine learning [2]. Amazon Web Services (AWS), one of the largest cloud providers in the world, offers instances with GPUs and FPGAs as well [3], indicating that such heterogeneous platforms are desirable for running contemporary and future computing applications.

Heterogeneous memory systems pose several challenges to system design and programmability. The complexity is exacerbated if such systems are built by separate vendors; for instance CPU–GPU platforms built with Intel CPUs and NVIDIA GPUs (e.g., P2 and G2 instances in AWS EC2 [3]). Furthermore, installed main memory capacity per machine is rapidly growing to catch up with the trend of large in-memory datasets for data-center/cloud applications [4].

Differences in memory characteristics and rapid growth in memory capacity make memory management of such heterogeneous systems challenging. Decisions about data placement and movement have to be made in an application-transparent manner while simplifying system design and programmability of already complex heterogeneous systems. Application-transparent policies are desirable as they do not require any extra effort from the programmer, which makes the adoption of such policies easier. Simple but effective designs can reduce development and verification effort, which pares down the time to market, enabling easier adoption of heterogeneous memories in production systems.

Previous academic and commercial systems studied Non-Uniform Memory Access (NUMA) extensively. The main optimization goal in such NUMA systems was to cut down on memory access latency as much as possible. However, in domains such as GPGPU computing or data-centers, characteristics such as bandwidth or access cost per bit are

far more critical. In this thesis, we study memory characteristics beyond memory access latency with the goal to simplify system design, easing programmability, while squeezing the best possible performance out of such heterogeneous memory systems. Next we briefly discuss key aspects that are relevant to modern incarnations of heterogeneous memory in GPUs and data-centers.

1. **Different memory bandwidths:** Different memory technologies will typically have differences in their bandwidths, e.g., a GPU connected to on-board GDDR and host-side DDR memories can have a bandwidth differential of $2 - 8\times$ between the two memory technologies (example systems shown in Figure 1.1). Since GPUs are sensitive to main memory bandwidth (due to massive thread-level parallelism), the metric to optimize in such systems is the total available bandwidth to the GPU from the two memory sources. While prior work on Non-Uniform Memory Access has closely looked at data placement strategies to optimize the total memory *access latency* in the presence of different memory zones with different access latencies, we show that such policies are not suited to optimizing the total memory *bandwidth*. Instead, we propose *Bandwidth-Aware Placement* (BW-AWARE), which directly maximizes the overall bandwidth, and show that it can significantly increase application throughput for several GPU applications.

2. **Different memory coherency domains:** In a heterogeneous CPU–GPU memory system, hardware cache coherence can improve performance by allowing concurrent, fine-grained access to memory by both CPU and GPU. However, implementing hardware cache coherence between the CPU and the GPU can lead to significant challenges because of design and verification involved particularly if the two domains are designed by separate vendors. We introduce *Selective Caching*, a coherence policy that disallows GPU caching of any memory that would require coherence updates to propagate across the two domains. This approach decouples the cache-coherence protocols in CPUs and GPUs, thus has a potential to improve cross-vendor design cycle time.

3. **Different costs per bit:** Different memory technologies can have different monetary costs. For example, recently announced non-volatile memory technology [5] is projected to be significantly cheaper than regular DRAM, while also being significantly higher latency. Its lower cost per bit makes it a good candidate for use in data-centers, where main memory cost can be $\approx$ 30% of the server machine cost, translating to $3 - 15\%$ of the total cost of ownership (TCO), amounting to billions of

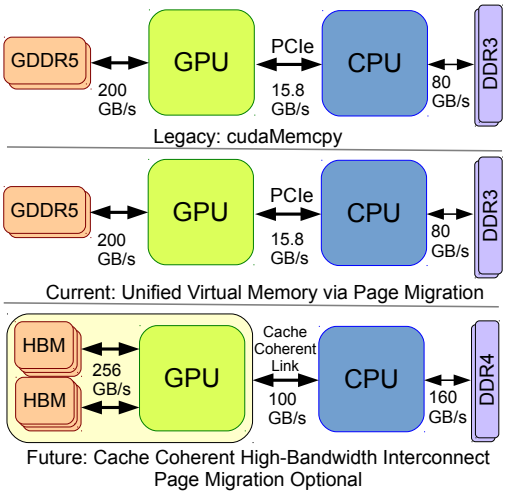


Figure 1.1: System architectures for legacy, current, and future mixed GPU-CPU systems.

dollars [6]. However, higher memory access latency means that only *cold*, i.e., infrequently accessed data can be placed in such memories. Detection of such cold data should be done *application-transparently* , since that allows cloud providers to transparently swap out slow memory in place of DRAM. We study how to place as much data in slow memory as possible, while being completely application-transparent, to improve performance per dollar in data-centers.

Below, we describe each of these three problems in detail and give a brief sketch of our proposed solution to these problems.

1.1 Bandwidth-asymmetric Systems

GPU-attached Bandwidth-Optimized (BO) memory (e.g., GDDR5) has been allocated and managed primarily through explicit, programmer-inserted function calls for obtaining maximum throughput out of GPUs. To make best use of the bandwidth available to GPU programs, programmers have to explicitly call memory copy functions to copy the data over the relatively slow PCIe bus to the GPU memory, and — only then — launch their GPU kernels. This up-front data allocation and transfer has been necessary since transferring data over the PCIe bus is a high overhead operation, and a bulk transfer of data amortizes this overhead. This data manipulation overhead also results in significant porting challenges when retargeting existing applications to GPUs, particularly for high-level languages that make use of libraries and dynamic memory allocation during application execution.

Recognizing the obstacle that explicit, programmer-inserted function call poses to the wider adoption of GPUs in more parallel applications, programming systems like NVIDIA's CUDA, OpenCL, and OpenACC are evolving to shared virtual address space between CPU and GPU [7]. With the availability of Unified Virtual Memory (UVM) [7] for NVIDIA GPUs the programmer-directed manual copy requirements has been lifted. Concurrently, CPU–GPU architectures are evolving to have unified globally addressable memory systems in which both the GPU and CPU can access any portion of memory at any time, regardless of its physical location. Today this unified view of memory is layered on top of legacy hardware designs by implementing software-based runtimes that dynamically copy data on demand between the GPU and CPU [8]. This on-demand copying results in significant throughput degradation as data copy time is not overlapped with GPU kernel launch latency, resulting in exposure of data copy latency to the GPU kernel run time. Thus, systems with the highest throughput requirements (i.e., majority of GPGPU systems) still use programmer-optimized code to run on GPUs.

As depicted in Figure 1.1, over the next several years it is expected that GPU and CPU systems will move away from the PCIe interface to a fully cache coherent (CC) interface [9]. These systems will provide high bandwidth ($5-10\times$ higher) and low latency ($10\times$ lower) between the non-uniform memory access (NUMA) pools attached to discrete processors by layering coherence protocols on top of physical link technologies such as NVLink [10], Hypertransport [11], or QPI [12].

As heterogeneous CPU–GPU systems move to a transparent unified memory system, the OS and runtime systems need information about other aspects of memory zones such as their bandwidths instead of only the access latency information that is exposed today via Advanced Configuration and Power Interface. In CC-NUMA systems today, latency information alone is adequate as CPUs are generally more performance sensitive to memory system latency rather than other memory characteristics. In contrast, massively parallel GPUs and their highly-threaded programming models have been designed to gracefully handle long memory latencies, instead demanding high bandwidth. Unfortunately, differences in bandwidth capabilities, read versus write performance, and access energy are not exposed to software; making it difficult for the operating system, runtime, or programmer to make good decisions about memory placement in these GPU-equipped systems. In this thesis (Chapters III, IV) we investigate the effect on GPU performance of exposing memory system bandwidth information to the operating system/runtime and user applications to improve the quality of dynamic page placement and migration decisions. We propose Bandwidth-Aware (BW-AWARE) page placement and dynamic page migration to fully utilize memory bandwidth for throughput-oriented CPU–GPU heteroge-

neous memory systems [13, 14].

1.2 Different Coherence Domains

Introducing globally visible shared memory improves programmer productivity by eliminating explicit copies and memory management overheads. Whereas this abstraction can be supported using only software page-level protection mechanisms [7, 15], hardware cache coherence can improve performance by allowing concurrent, fine-grained access to memory by both CPU and GPU.

Despite the programmability benefits of CPU–GPU cache coherence, designing such a system can pose several hurdles. Prior studies [16, 17] have shown that coherence implementations are a major source of hardware design bugs. Extending a CPU coherence implementation to a GPU over a long-latency interconnect ($\approx$ 100ns) will only increase the design cost of such a system. Also, if the CPUs and GPUs are to be manufactured by different vendors, a high level of coordination is needed between those two vendors – including coordination on the specification of coherence implementation and verification efforts. Quoting George Bernard Shaw: *"The single biggest problem in communication is the illusion it has already taken place."* Such hurdles make CPU–GPU cache coherence an unattractive option to deploy in a product.

Current CPUs have up to 18 cores per socket [18] but GPUs are expected to have hundreds of streaming multiprocessors (SMs) each with its own cache(s) within the next few years. Hence, extending traditional hardware cache-coherency into a multi-chip CPU–GPU memory system requires coherence messages to be exchanged not just within the GPU but over the CPU–GPU interconnect. Keeping these hundreds of caches coherent with a traditional HW coherence protocol, as shown in Figure 6.1, potentially requires large state and interconnect bandwidth [19, 20]. Some recent proposals call for data-race-free GPU programming models, which allow relaxed or scoped memory consistency to reduce the frequency or hide the latency of enforcing coherence [21]. However, irrespective of memory ordering requirements, such approaches still rely on hardware cache-coherence mechanisms to avoid the need for software to explicitly track and flush modified cache lines to an appropriate scope at each synchronization point. Techniques like region coherence [22] seek to scale coherence protocols for heterogeneous systems, but require pervasive changes throughout the CPU and GPU memory systems. Such approaches also incur highly coordinated design and verification effort by both CPU and GPU vendors [16] that is challenging when multiple vendors wish to integrate existing CPU and GPU designs in a timely manner.

Due to the significant challenges associated with building such cache-coherent systems, in this thesis (Chapter V), we architect a GPU *selective caching* mechanism [23]. This mechanism provides the conceptual simplicity of CPU–GPU hardware cache coherence and maintains a high level of GPU performance (93% of hardware cache-coherent system), but does not actually implement complex hardware cache coherence between the CPU and GPU.

1.3 Cheaper Memory Technologies

Upcoming memory technologies, such as Intel/Micron's recently-announced 3D XPoint memory [24], are projected to be denser and cheaper per bit than DRAM while providing the byte-addressable load-store interface of conventional main memory. Improved capacity and cost per bit come at the price of higher access latency, projected to fall somewhere in the range of 400ns to several microseconds [24] as opposed to 50–100ns for DRAM. Slow memory can result in a net cost win if the cost savings of replaced DRAM outweigh the cost increase due to reduced program performance or by enabling a higher peak memory capacity per server than is economically viable with DRAM alone.

However, deploying such memories in a cloud service for use by its customers is riddled with several challenges. It is important for cloud providers to be able to provide service-level-agreements (SLAs) to customers guaranteeing performance of the cloud platform. To provide such SLAs in the presence of slow memory, any memory placement policy must estimate the performance degradation associated with placing a given memory page in slow memory, which in turn requires some method to gauge the *page access rate*. Lack of accurate *and* performant page access rate tracking in contemporary x86 hardware makes this task challenging.

Making slow memory use application-transparent is particularly critical for cloud computing environments, where the cloud provider may wish to transparently substitute cheap memory for DRAM to reduce provisioning cost, but has limited visibility and control of customer applications. Relatively few cloud customers are likely to take advantage of cheaper-but-slower memory technology if they must redesign their applications to explicitly allocate and manage hot and cold memory footprints. A host-OS-based cold memory detection and placement mechanism is a natural candidate for such an environment.

In this thesis (Chapter VI) we propose *Thermostat* [25] that manages two-tiered main memory transparently to applications while preserving the benefits of huge pages and dynamically enforcing limits on performance degradation (e.g., limiting slowdown to 3%).

1.4 Contributions

In this thesis we make following contributions:

- We show that existing CPU-oriented page placement policies are sub-optimal for placement in bandwidth-asymmetric GPU-based memory systems and do not have enough information to make informed decisions when optimizing for GPU applications that run thousands of threads launching multiple memory requests in parallel. We show that placing all pages in the highest bandwidth memory is not the best performing page placement policy for GPU workloads and propose a new bandwidth-aware (BW-AWARE) page placement policy that can concurrently maximize the bandwidth available from different memory technologies. (Chapter III)

- In regards to page migration policies, we show that counter-based metrics to determine when to migrate pages from the CPU to GPU are insufficient for finding an optimal migration policy to exploit GPU memory bandwidth. In streaming workloads, where each page may be accessed only a few times, waiting for N accesses to occur before migrating a page will actually limit the number of accesses that occur after migration, reducing the efficacy of the page migration operation. We propose a page migration policy which is aware of bandwidth differential and balances migration with CC-NUMA link utilization that outperforms either CPU or GPU memory being used in isolation. (Chapter IV)

- We propose GPU selective caching, which can provide a CPU–GPU system that provides a unified shared memory without requiring hardware cache-coherence protocols within the GPU or between CPU and GPU caches. We identify that much of the improvement from GPU caches is due to the request filtering to lower memory hierarchy. We capture spatial locality by coalescing requests in miss status handling registers (MSHRs) and propose a small on-die CPU cache specifically to handle uncached requests that will be issued at sub-cache line granularity from the GPU. This cache helps both shield the CPU memory system from the bandwidth hungry GPU and supports improved CPU–GPU interconnect efficiency by implementing variable-sized transfer granularity. (Chapter V)

- To reduce cost of memory in large scale data-centers and cloud platforms, we make a case for using upcoming cheaper but slower memory technologies, such as Intel's/Micron's 3D XPoint. We propose Thermostat, an online low-overhead application-transparent page management mechanism for controlling performance

degradation due to placing infrequently accessed pages (cold pages) in slow memory for two-tiered main memory system. Thermostat supports huge pages, which are performance critical to applications running on cloud platforms under virtualization. With Thermostat, cloud vendors can move infrequently accessed pages to cheaper/slower memory and keep the application slowdown in control meeting customer's throughput demands. By moving pages to cheaper memory, DRAM's capacity can be freed up to pack more virtual machines on the same machine reducing the amount of machines required for large footprint applications, thereby reducing the cost reflected on cloud customers. (Chapter VI)

1.5 Impact

Simple but effective page management algorithms have enormous sustainability, with no major changes to locality based page placement occurring in the last several decades. With the introduction of high bandwidth memory, heterogeneous memory CPU + GPU systems started shipping in 2016, yet operating systems today are ill prepared to take advantage of performance-asymmetric memories. As these systems proliferate, OS page placement policies must fundamentally be re-architected to account for the underlying topology and technology differences that will be present in many, if not most, memory systems. While latency-bound applications (typically run on the CPU) will likely continue to use locality (and thus latency) to drive their page placement decisions, bandwidth-bound applications (like GPU workloads) cannot maximize performance without the introduction of new BW-AWARE page placement algorithms. Similarly, without appropriate page placement strategies the energy efficiency opportunities presented by heterogeneous memories will not be realized.

BW-AWARE page placement and application aware page placement works seamlessly across systems with varying memory bandwidth ratios, because the underlying operating system and runtime environment actively query the available bandwidth of hardware at execution time. On the other hand, approaches like hardware DRAM cache approaches are fixed at runtime and optimal cache design is typically a function of cache capacity to backing memory capacity as well as the relative bandwidths between the high bandwidth cache memory and backing memory bandwidth. The combination of conceptual and design simplicity makes BW-AWARE page placement likely to be used in most systems, cementing its place as the standard management policy for heterogeneous memories, for many years to come.

Programmability with Performance: Whereas cache-coherent access can boost

programmer productivity significantly, it is difficult to implement cache coherent memory for discrete GPU systems, particularly when the CPU and GPU are supplied by different vendors (e.g., NVIDIA GPUs in IBM Power servers). Selective caching eliminates the need for a unified hardware cache coherence protocol, removing the most difficult aspect of providing a unified shared memory implementation. We see selective caching occupying a feasibility sweet spot; sacrificing a small amount of performance for enormous wins in conceptual and implementation complexity.

Verification and Time-to-market: Verification is a major factor in the total cost and time-to-market to ship a processor. CPU–GPU coherence increases verification overhead significantly, since it requires a combined validation of the CPU and GPU designs. Debugging problems are particularly difficult in multi-vendor collaborations, because each vendor is incentivized to reveal as little of their intellectual property as possible to their counterparts to preserve market advantage. Selective caching simplifies such collaborations by removing tight technical entanglements between designs, which can significantly reduce the time-to-market of new products.

Because it is difficult to quantify, design simplicity is rarely a focus of academic research. Yet, we argue that hidden complexities, such a validation efforts or the requirement for industry wide coordination are often key impediments that prevents practical adoption of otherwise laudable academic proposals. We have partnered with product groups both within NVIDIA and IBM while pursuing this research to ensure that practical implementation concerns are first and foremost throughout our research [26].

Cheaper Memory Technology in Cloud Deployments: Slower but cheaper memory technologies such as NVRAMs hold great promise to lower the memory costs for cloud vendors. However, vendors have little control over the client applications, and have to guarantee Service Level Agreements. Also, dependence on hardware features only delays the adoption of such technologies due to the time required for rolling out new features in CPUs. Thermostat is the first policy to our knowledge that addresses these challenges by a software-only mechanism, thus paving the way towards large-scale deployments of such technologies in datacenters.

Also, Thermostat's novel software-only sampling policy shows the potential for using software-only mechanisms for inferring application memory patterns in runtime at extremely low overhead ($< 1\%$). We expect that Thermostat's sampling policy can be employed in other scenarios that require runtime monitoring of application access patterns, such as resource sharing between applications, memory access profile-guided optimizations etc.

CHAPTER II

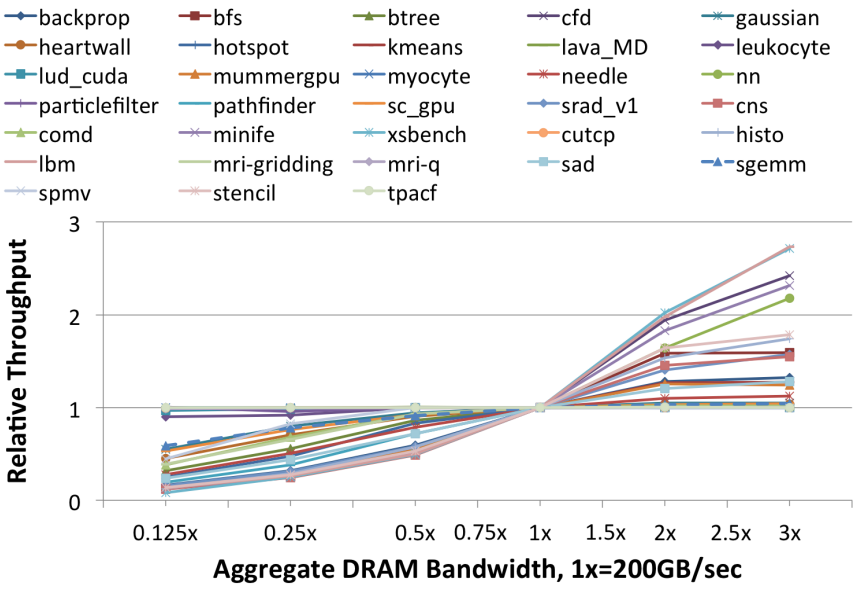
Background and Motivation

Systems using heterogeneous CPU and GPU computing resources have been widely used for several years. High performance GPUs have developed into stand-alone PCIe-attached accelerators requiring explicit memory management by the programmer to control data transfers into the GPU's high-bandwidth locally attached memory. As GPUs have evolved, the onus of explicit memory management has been addressed by providing a unified shared memory address space between the GPU and CPU [7, 15]. Whereas a single unified virtual address space improves programmer productivity, discrete GPU and CPU systems still have separate locally attached physical memories, optimized for bandwidth and latency respectively.

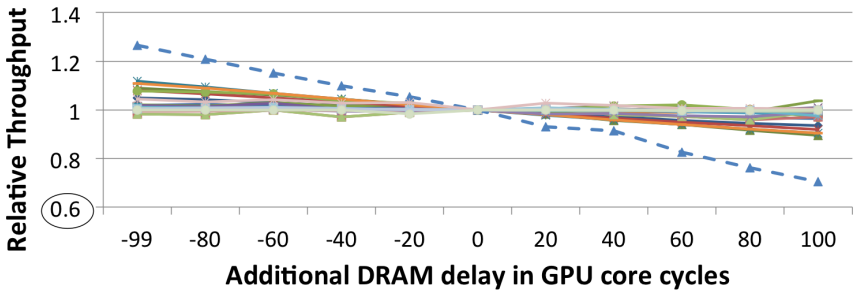
2.1 Bandwidth Hungry Characteristics of GPUs

A by-product of the GPU's many-threaded design is that it is able to maintain a large number of in-flight memory requests and execution throughput is correlated to memory bandwidth rather than latency, as compared to CPU designs. As a result, GPUs have chosen to integrate high bandwidth off-package memory like GDDR rather than accessing the CPU's DDR directly or integrating DDR locally on the GPU board.

To highlight the sensitivity of GPU performance to memory characteristics, Figures 2.1a and 2.1b show the performance variation as memory bandwidth and latency vary for a variety of GPU compute benchmarks from the Rodinia [27] and Parboil [28] suites, as well as a number of recent HPC [29, 30, 31, 32] workloads. Most of these GPU workloads are sensitive to changes in bandwidth, while showing much more modest sensitivity to varying the latency; only `sgemm` stands out as highly latency sensitive among these 33 workloads. Some application kernels are neither bandwidth nor latency sensitive and do not see significant performance variation as modifications are made to the memory sub-



(a) Bandwidth sensitivity



(b) Latency sensitivity

Figure 2.1: GPU performance sensitivity to bandwidth and latency changes.

system. While GPU-equipped systems generally require bandwidth-optimized memories to achieve peak performance, these memory technologies have significant cost, capacity, and/or energy disadvantages over alternative DRAM technologies.

The most common Bandwidth-Optimized (BO) memory technology today is GDDR5 [33]. Providing a per-pin data rate of up to 7Gbps, this memory technology is widely used with discrete GPUs used in HPC, workstation, and desktop systems. Due to the high data rates, GDDR5 systems require significant energy per access and are unable to support high-capacity multi-rank systems. In contrast, the roadmap for the next several years of cost/capacity-optimized (CO) DRAM (DDR4 and LPDDR4) provides a per-pin data rate that reaches only 3.2 Gbps. However, these CO DRAM technologies provide similar latency at a fraction of the cost and lower energy per access compared to the BO GDDR5 memories. Looking forward, systems requiring more bandwidth and/or re-

duced energy per access are moving to die-stacked DRAM technologies [34, 35]. These bandwidth-optimized stacked memories are significantly more energy-efficient than off-package memory technologies like GDDR5, DDR4, and LPDDR4. Unfortunately, the number of DRAM die that can be economically stacked in a single package is limited, necessitating systems to also provide a pool of off-package capacity-optimized DRAM.

This disaggregation of memory into on-package and off-package pools is one factor motivating the need to revisit page placement within the context of GPU performance. Future GPU/CPU systems are likely to take this disaggregation further and move capacity-optimized memory not just off the GPU package, but across a high speed interconnect where it is physically attached to the CPU rather than the GPU, or possibly further [36]. In a CC-NUMA system, the physical location of this capacity-optimized memory only changes the latency and bandwidth properties of this memory pool – it is functionally equivalent regardless of being CPU or GPU locally attached. A robust page placement policy for GPUs will abstract the on-package, off-package, and remote memory properties into performance and power characteristics based on which it can make optimized decisions.

2.2 Current OS NUMA Page Placement

In modern symmetric multiprocessor (SMP) systems, each socket typically consists of several cores within a chip multi-processor (CMP) that share last-level caches and on-chip memory controllers [37]. The number of memory channels connected to a processor socket is often limited by the available pin count. To increase the available memory bandwidth and capacity in a system, individual sockets can be connected via a cache coherent interconnect fabric such as Intel's Quick Path [12], AMD's HyperTransport [11], or NVIDIA's NVLink [10]. A single socket, the processors within it, and the physically attached memory comprise what an operating system sees as a local NUMA zone. Each socket is a separate NUMA zone. While a processor within any given zone can access the DRAM within any other zone, there is additional latency to service this memory request compared to a locally serviced memory request because the request must be routed first to its own memory controller, across the socket interconnect, and through the remote memory controller.

Operating systems such as Linux have recognized that, unless necessary, it is typically better for applications to service memory requests from their own NUMA zone to minimize memory latency. To get the best performance out of these NUMA systems, Linux learns system topology information from the Advanced Configuration and Power Interface (ACPI)

System Resource Affinity Table (SRAT) and memory latency information from the ACPI System Locality Information Table (SLIT). After discovering this information, Linux provides two basic page placement policies that can be specified by applications to indicate where they prefer their physical memory pages to be placed when using standard `malloc` and `mmap` calls to allocate memory.

LOCAL: The default policy inherited by user processes is *LOCAL* in which physical page allocations will be from memory within the local NUMA zone of the executing process, unless otherwise specified or due to capacity limitations. This typically results in allocations from memory physically attached to the CPU on which the process is running, thus minimizing memory access latency.

INTERLEAVE: The second available allocation policy, which processes must specifically inform the OS they would like to use, is *INTERLEAVE*. This policy allocates pages round-robin across all (or a subset) of the NUMA zones within the SMP system to balance bandwidth across the memory pools. The downside of this policy is that the additional bandwidth comes at the expense of increased memory latency. Today, the OS has no knowledge about the relative bandwidth of memories in these different NUMA zones because SMP systems have traditionally had bandwidth-symmetric memory systems.

In addition to these OS placement policies, Linux provides a library interface called *libNUMA* for applications to request memory allocations from specific NUMA zones. This facility provides low-level control over memory placement but requires careful programming because applications running on different systems will often have different NUMA-zone layouts. Additional difficulties arise because there is no performance feedback mechanism available to programmers when making memory placement decisions, nor are they aware of which processor(s) their application will be running on while writing their application.

With the advent of heterogeneous memory systems, the assumptions that operating system NUMA zones will be symmetric in bandwidth and power characteristics break down. The addition of heterogeneous GPU and CPU computing resources further stresses the page placement policies since processes may not necessarily be migrated to help mitigate performance imbalance, as certain phases of computation are now pinned to the type of processor executing the program. As a result, data placement policies combined with bandwidth-asymmetric memories can have significant impact on GPU, and possibly CPU, performance.

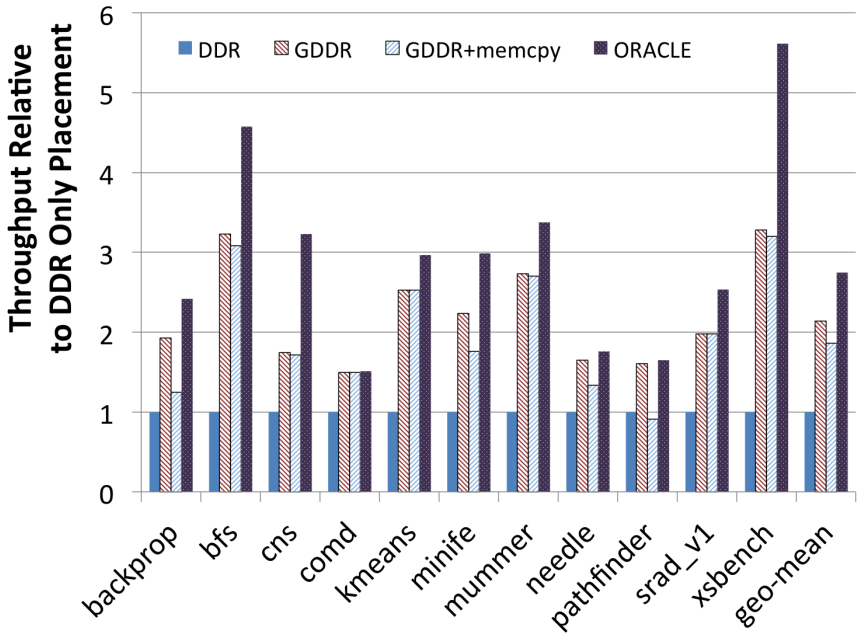


Figure 2.2: GPU performance sensitivity to memory subsystem performance where GDDR provides 200GB/s, DDR provides 80GB/s, and `memcpy` bandwidth is 80GB/s.

2.3 Memory Copy Overhead in GPUs

In current CPU/GPU designs, GPU and CPU memory systems are private and require explicit copying to the GPU before the application can execute. Figure 2.2 shows the effect of this copy overhead on application performance by comparing GDDR to GDDR+`memcpy` performance which includes the cost of the programmer manually copying data from the DDR to the GDDR before launching the GPU kernels. While this copy overhead varies from application to application, it can be a non-trivial performance overhead for short-running GPU applications and can even negate the effectiveness of using the high bandwidth GDDR on-board the GPU in a limited number of cases.

While it is technically possible for the GPU to access CPU memory directly over PCIe today, the long latency (microseconds) of the access makes this a rarely used memory operation. Programming system advancements enabling a uniform global address space, like those introduced in CUDA 6.0 [8], relax the requirement forcing programmers to allocate and explicitly copy memory to the GPU up-front, but do nothing to improve the overhead of this data transfer. Further, by copying pages from the CPU to the GPU piecemeal on demand, these new runtimes often introduce additional overhead compared to performing a highly optimized bulk transfer of all the data that the GPU will need during execution. The next step in the evolution of GPUs, given the unified addressing, is to

optimize the performance of this new programming model.

2.4 Cache Coherent GPUs Enhancing System Programmability

The key advancement expected to enable performance is the introduction of CC-NUMA GPU and CPU systems. Using cache coherence layered upon NVLink, HT, or QPI, GPUs will likely be able to access CPU memory in hundreds of nanoseconds at bandwidths up to 128GB/s by bringing cache lines directly into GPU caches. Figure 2.2 shows the upper bound (labeled ORACLE) on performance that could be achieved if both the system DDR memory and GPU GDDR memory were used concurrently, assuming data had been optimally placed in both technologies. In this work, we define oracle placement to be *a priori* page placement in the GPU memory (thus requiring no migration), of the minimum number of pages, when sorted from hottest to coldest, such that the GDDR bandwidth is fully subscribed during application execution.

Because initial CPU/GPU CC-NUMA systems are likely to use a form of IOMMU address translation services for walking the OS page tables within the GPU, it is unlikely that GPUs will be able to directly allocate and map their own physical memory without a call back to the CPU and operating system. In this work, we make a baseline assumption that all physically allocated pages are initially allocated in the CPU memory and only the operating system or GPU runtime system executing on the host can initiate page migrations to the GPU. In such a system, two clear performance goals become evident. The first is to design a memory policy that balances CC-NUMA access and page migration to simply achieve the performance of the legacy bulk copy interface without the programming limitations. The second, more ambitious, goal is to exceed this performance and approach the oracular performance by using these memory zones concurrently, enabling a peak memory bandwidth that is the sum of the two zones.

Achieving either of these goals requires migrating enough data to the GPU to exploit its high memory bandwidth while avoiding migrating pages that may never be accessed again. Every page migration increases the total bandwidth requirement of the application and over-migration potentially reduces application performance if sufficient bandwidth headroom in both the DDR and GDDR is not available. Thus, the runtime system must be selective about which pages to migrate. The runtime system also must be cognizant that performing TLB invalidations (an integral part of page migration) on a GPU does not just halt a single processor, but thousands of compute pipelines that may be accessing these pages through a large shared TLB structure. This shared TLB structure makes page migrations between a CPU and GPU potentially much more costly (in terms of the

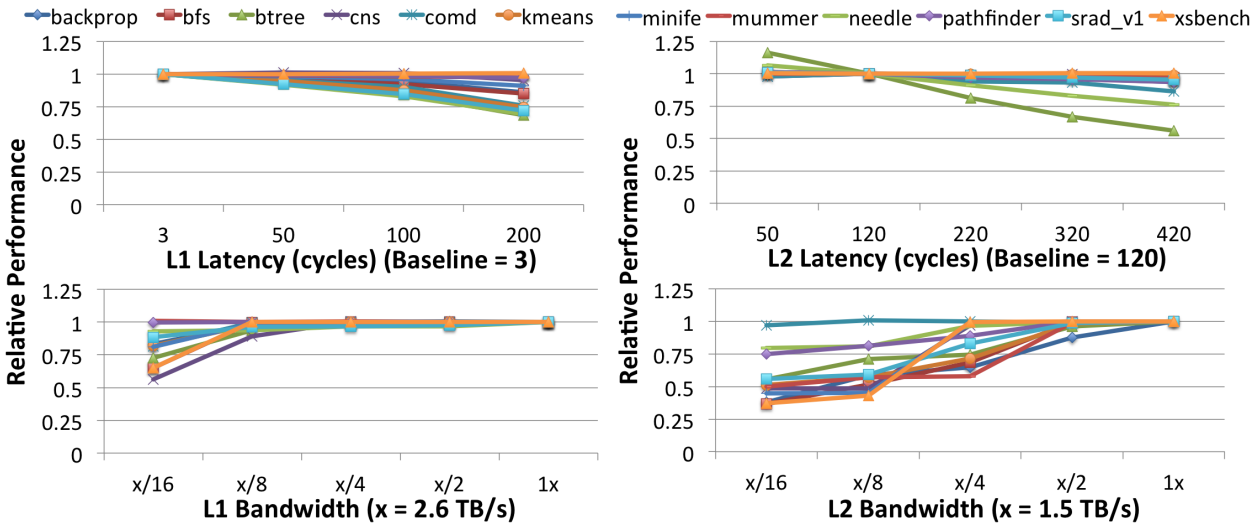


Figure 2.3: GPU performance sensitivity to L1 and L2 latency and bandwidth changes.

opportunity cost of lost execution throughput) than in CPU-only systems.

In addition to managing the memory bandwidth overhead of page migration and execution stalls due to TLB shootdowns, the relative bandwidth utilization of both the CPU and GPU memory must be taken into account when making page migration decisions. When trying to balance memory bandwidth between two distinct memory zones, it is possible to over- or under-subscribe either memory zone. Migrating pages too slowly to the GPU memory will leave its local memory sitting idle, wasting precious bandwidth. Conversely, migrating pages to the GPU too aggressively may result in under-utilization of the CPU memory while paying the maximum cost in terms of migration overheads. A comprehensive CPU-GPU memory management solution will attempt to balance all of these effects to maximize memory system and GPU throughput in future mobile, graphics, HPC, and data-center installations.

2.5 Supporting Hardware Cache Coherence in GPUs

Managing the physical location of data, and guaranteeing that reads access the most up-to-date copies of data in a unified shared memory can be done through the use of page level migration and protection. Such mechanisms move data at the OS page granularity between physical memories [7]. With the advent of non-PCIe high bandwidth, low latency CPU–GPU interconnects, the possibility of performing cache-line, rather than OS-page-granularity, accesses becomes feasible. Without OS level page protection mechanisms to support correctness guarantees, however, the responsibility of coherence has typically fallen on hardware cache-coherence implementations.

Workload	L1 Hit Rate (%)	L2 Hit Rate (%)
backprop	62.4	70.0
bfs	19.6	58.6
btree	81.8	61.8
cns	47.0	55.2
comd	62.5	97.1
kmeans	5.6	29.5
minife	46.7	20.4
mummer	60.0	30.0
needle	7.0	55.7
pathfinder	42.4	23.0
srad_v1	46.9	25.9
xsbench	30.7	63.0
Arith Mean	44.4	51.6

Table 2.1: GPU L1 and L2 cache hit rates (average).

As programming models supporting transparent CPU-GPU sharing become more prevalent and sharing becomes more fine-grain and frequent, the performance gap between page-level coherence and fine-grained hardware cache-coherent access will grow [14, 13, 38]. On-chip caches, and thus HW cache coherence, are widely used in CPUs because they provide substantial memory bandwidth and latency improvements [39]. Building scalable, high-performance cache coherence requires a holistic system that strikes a balance between directory storage overhead, cache probe bandwidth, and application characteristics [22, 40, 41, 20, 16, 42, 19]. Although relaxed or scoped consistency models allow coherence operations to be re-ordered or deferred, hiding latency, they do not obviate the need for HW cache coherence. However, supporting a CPU-like HW coherence model in large GPUs, where many applications do not require coherence, is a tax on GPU designers. Similarly, requiring CPUs to relax or change their HW coherence implementations or implement instructions enabling software management of the cache hierarchy adds significant system complexity.

Prior work has shown that due to their many threaded design, GPUs are insensitive to off-package memory latency but very sensitive to off-chip memory bandwidth [14, 13]. Table 2.1 shows the L1 and L2 cache hit rates across a variety of workloads from the Rodinia and United States Department of Energy application suites [27, 43]. These low hit rates cause GPUs to also be fairly insensitive to small changes in L1 and L2 cache latency and bandwidth, as shown in Figure 2.3. This lack of sensitivity raises the question whether GPUs need to uniformly employ on-chip caching of all off-chip memory in order to achieve good performance. If GPUs do not need or can selectively employ on-chip caching, then

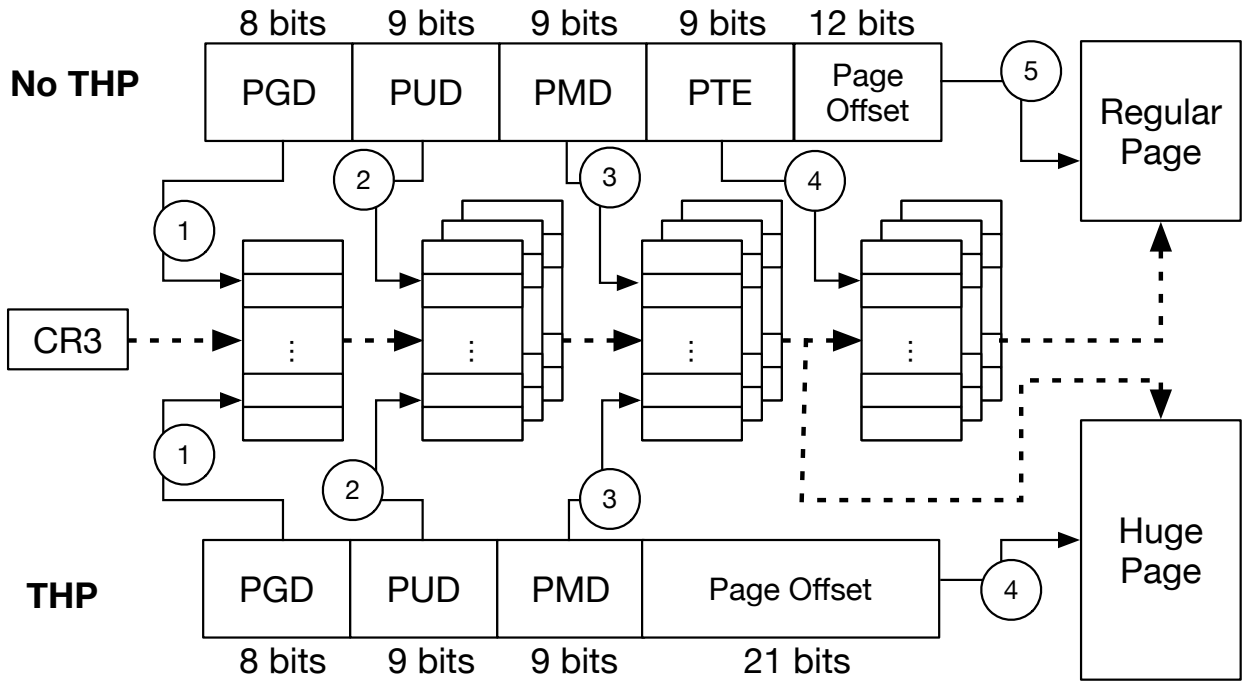


Figure 2.4: Linux page table structure for X86 both with and without a transparent huge page.

CPU–GPU systems can be built that present a unified coherent shared memory address space to the CPU, while not requiring a HW cache-coherence implementation within the GPU.

Avoiding hardware cache coherence benefits GPUs by decoupling them from the coherence protocol implemented within the CPU complex, enables simplified GPU designs, and improves compatibility across future systems. It also reduces the scaling load on the existing CPU coherence and directory structures by eliminating the potential addition of hundreds of additional caches, all of which may be sharing data. Selective caching does not come without a cost however. Some portions of the global memory space will become un-cacheable within the GPU and bypassing on-chip caches can place additional load on limited off-chip memory resources. In the following sections, we show that by leveraging memory request coalescing, small CPU-side caches, improved interconnect efficiency, and promiscuous read-only caching, selective caching GPUs can perform nearly as well as HW cache-coherent CPU–GPU systems.

2.6 Virtual Memory Management in Linux

As the size of main memory has grown, the overheads of virtual memory systems have grown as well. The hierarchical Linux page table for the x86-64 architecture, shown in Figure 2.4, is four levels – and thus may incur up to four extra main memory accesses to walk the page table in [44]. Moreover, execution in virtualized environments can increase number of memory accesses to 24, a $6x$ increase from the non-virtualized case [45, 46, 47]. As a result, the Translation Lookaside Buffer (TLB), which acts as a cache for virtual-to-physical mappings, has become increasingly important to mollify the effects of virtual memory translation overhead.

In most architectures, TLB accesses lie on the critical path of memory accesses, hence hardware timing constraints limit the number of TLB entries that can be searched on each access. As memory capacities grow while page sizes remain constant, TLB coverage – the fraction of main memory that can be represented by the contents of the TLB at any given time – necessarily decreases. This reduced coverage hampers the performance of programs with large working sets and/or instruction footprints because it increases the number of TLB misses they exhibit, and thus the number of page table walks that must be performed. What's more, as working set sizes – and, correspondingly, page table sizes – grow, the fraction of translation data that can fit in the cache is reduced, leading to more severe TLB miss penalties.

Huge pages directly address the costs of fine-grain page management. They effectively increase TLB coverage – one huge page TLB entry covers the area of a large number of regular page entries – thus reducing page faults, and they reduce the size of the page table structure, leading to both fewer memory accesses upon a TLB miss and increased cache-ability of translation data.

2.7 Transparent Huge Pages

Early system support for huge pages, static huge pages, requires bucketing the available physical memory at boot time into standard pages and huge pages. Static huge pages are disjoint from the standard memory pool and cannot be used for the disk page cache, disk write buffers, or any kernel data structure. Moreover static huge pages require application changes and are thus non-transparent to the programmer. The static provisioning of memory into static huge pages also leads to allocation problems and memory stranding if the actual demand for huge pages does not match the boot-time configuration. Thus, even if applications are static huge page-aware, such a solution is less than

ideal for many systems because it requires a priori knowledge of applications' memory requirements to ensure ideal performance.

Recent versions of the Linux kernel instead exploit huge pages through THP. With THP, the kernel attempts to invisibly (i.e., without the knowledge of the user) allocate huge pages to back large regions of contiguous virtual memory. Transparent allocation is advantageous because it allows existing code bases to reap the rewards of huge pages without modification and doesn't require changing the interfaces and invariants of system calls and functions that require consideration of page size, such as `mmap()`. However, the kernel may demote allocated huge pages by breaking them down into a set of regular pages when it deems necessary to maintain support for functions that are not huge page-aware.

Allocation/Promotion: The first time an application touches an allocated region of virtual memory, a page fault occurs and the kernel allocates one or more physical pages to back that region and records the virtual to physical mapping. With THP enabled, the kernel allocates huge pages for anonymous (i.e., non-file-backed) regions of huge page-sized and -aligned virtually contiguous memory during the initial page fault to that region. Alternatively, if a region isn't initially backed by a huge page, it can later undergo promotion, in which multiple regular pages are marked to be treated as a single huge page. The kernel can be set to either aggressively allocate huge pages whenever it can, or to only allocate them when the user provides hints via the `madvise()` system call.

Demotion: Any region backed by a huge page may be subject to spontaneous demotion to regular pages. Because various parts of the kernel source code are not huge page-aware, giving them access to a huge page could lead to unspecified or erroneous behavior. As a result, huge pages are often *split*, or broken into several regular pages, before they are passed to functions that are not huge page-aware. To perform this split, the page table must be updated to reflect the many regular pages that comprise the demoted huge page.

CHAPTER III

Page Placement Strategies for GPUs within Heterogeneous Memory Systems

3.1 Introduction

GPUs are now ubiquitous in systems ranging from mobile phones to data-centers like Amazon's elastic compute cloud (EC2) and HPC installations like Oak Ridge National Laboratory's Titan supercomputer. Figure 3.1 shows several processor and memory topology options that are likely to be common over the next several years. While traditional systems are likely to continue using commodity DDR3 and soon DDR4, many future GPUs and mobile systems are moving to also include higher bandwidth, but capacity limited, on-package memories such as High Bandwidth Memory (HBM) or Wide-IO2 (WIO2). Regardless the type of machine, both memories will be globally accessible to maximize aggregate capacity and performance, making all systems non-uniform memory access (NUMA) due to variations in latency, bandwidth, and connection topology. Depending on the memories paired the bandwidth ratio between the bandwidth-optimized (BO) and capacity or cost optimized (CO) memory pools may be as low as 2$\times$ or as high as 8$\times$.

Massively parallel GPUs and their highly-threaded programming models can tolerate long memory latencies but, these throughput oriented processors demand high bandwidth. However, due to lack of exposure of difference in bandwidth capabilities differential of main memory technologies OS or the programmer, they cannot make the best memory management decisions to exploit the memory bandwidth of heterogeneous CPU-GPU system. In this thesis we explore the effect on GPU performance of exposing memory system bandwidth information to the operating system/runtime and user applications to improve the quality of dynamic page placement decisions.

We explore two OS page placement policies:

1) Application agnostic Bandwidth-Aware (BW-AWARE) page placement policy that can

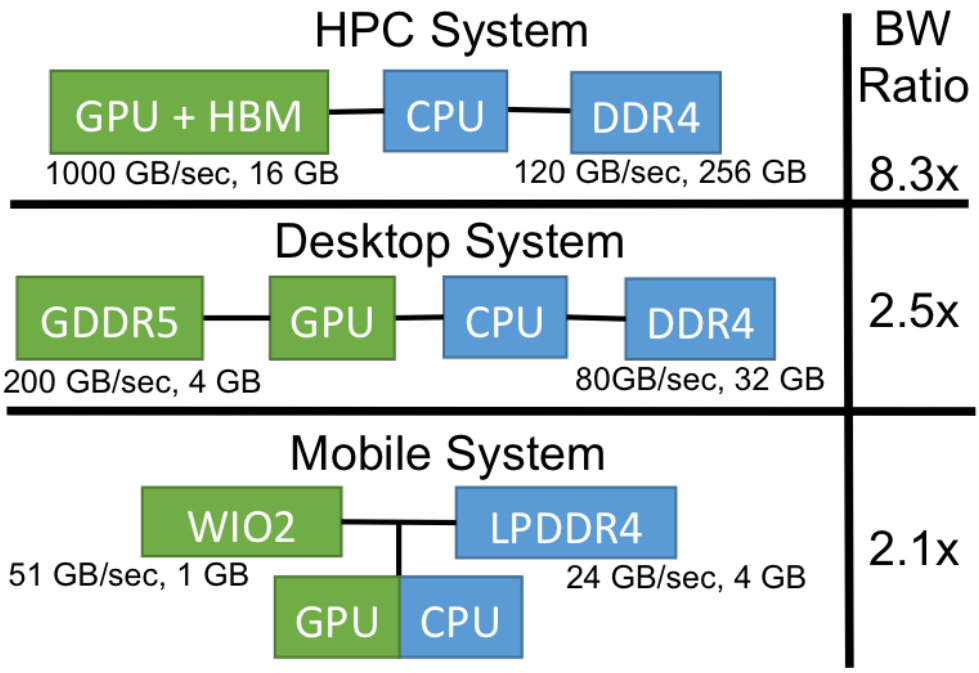


Figure 3.1: BW-Ratio of high-bandwidth vs high-capacity memories for likely future HPC, desktop, and mobile systems

outperform Linux's current bandwidth-optimized INTERLEAVE placement by 35% and the default latency optimized LOCAL allocation policy by as much as 18%, when the application footprint fits within bandwidth-optimized memory capacity.
2) For *memory capacity constrained* systems (i.e. bandwidth-optimized memory capacity is insufficient for the workload footprint), we demonstrate that using simple application annotations to inform the OS/runtime of hot versus cold data structures can outperform the current Linux INTERLEAVE and LOCAL page placement policies. Our annotation based policy combined with bandwidth information can outperform these page placement policies by 19% and 12% respectively, and get within 90% of oracle page placement performance.

3.2 BW-AWARE Page Placement

Using all available memory bandwidth is a key driver to maximizing performance for many GPU workloads. To exploit this observation, we propose a new OS page placement algorithm which accounts for the bandwidth differential between different bandwidth-optimized and capacity-optimized memories when making page placement decisions. This section discusses the need, implementation, and results for a bandwidth-aware (BW-

AWARE) page placement policy for systems where the application footprint fits within BO memory, the common case for GPU workloads today. Later in Section 3.4, we discuss an extension to BW-AWARE placement for systems where memory placement decisions are constrained by the capacity of the bandwidth-optimized memory. Both HPC systems trying to maximize in-memory problem footprint and mobile systems which are capacity limited by cost and physical part dimensions may soon encounter these capacity constraints with heterogeneous memories.

3.2.1 Bandwidth Maximized Page Placement

The goal of bandwidth-aware page placement is to enable a GPU to effectively use the total combined bandwidth of *all* the memory in the system. Because GPUs are able to hide high memory latency without stalling their pipelines, all memories in a system can be used to service GPU requests, even when those memories are off-package or require one or more hops through a system interconnect to access. To exploit bandwidth-heterogeneous memories, our BW-AWARE policy places physical memory pages in the ratio of aggregate bandwidths of the memories in the system without requiring any knowledge of page access frequency. Below we derive that this placement policy is optimal for maximizing bandwidth.

Consider a system with bandwidth-optimized and capacity-optimized memories with bandwidths b_B and b_C respectively, where unrestricted capacity of both memories are available. Let f_B represent fraction of data placed in the BO memory and $1 - f_B$ in the CO memory. Let us assume there are total of N memory accesses uniformly spread among different pages. Then the total amount of time spent by the BO memory to serve $N * f_B$ memory accesses is $N * f_B/b_B$ and that by the CO memory to serve $N(1 - f_B)$ memory accesses is $N(1 - f_B)/b_C$. Since requests to these two memories are serviced in parallel, the total time T to serve the memory requests is:

$$T = max(N * f_B/b_B, N(1 - f_B)/b_C)$$

To maximize performance, T must be minimized. Since, $N * f_B/b_B$ and $N(1 - f_B)/b_C$ are linear in f_b and $N * f_B/b_B$ is increasing function while $N(1 - f_B)/b_C$ is decreasing, the minimum of T occurs when both are equal:

$$T_{opt} = N * f_B/b_B = N(1 - f_B)/b_C$$

Simulator	GPGPU-Sim 3.x
GPU Arch	NVIDIA GTX-480 Fermi-like
GPU Cores	15 SMs @ 1.4Ghz
L1 Caches	16kB/SM
L2 Caches	Memory Side 128kB/DRAM Channel
L2 MSHRs	128 Entries/L2 Slice
Memory system	
GPU-Local GDDR5	8-channels, 200GB/sec aggregate
GPU-Remote DDR4	4-channels, 80GB/sec aggregate
DRAM Timings	RCD=RP=12,RC=40,CL=WR=12
GPU-CPU Interconnect Latency	100 GPU core cycles

Table 3.1: System configuration for heterogeneous CPU-GPU system.

Therefore,

$$f_{Bopt} = b_B/(b_B + b_C)$$

Because we have assumed that all pages are accessed uniformly, the optimal page placement ratio is the same as the bandwidth service ratio between the bandwidth-optimized and capacity-optimized memory pools. From this derivation we make two additional observations. First, BW-AWARE placement will generalize to an optimal policy where there are more than two technologies by placing pages in the bandwidth ratio of all memory pools. Second, a practical implementation of a BW-AWARE policy must be aware of the bandwidth provided by the various memory pools available within a system. Hence there is a need for a new System Bandwidth Information Table (SBIT), much like there is already a ACPI System Locality Information Table (SLIT) which exposes memory latency information to the operating system today. We will re-visit the assumption of uniform page access later in Section 3.3.1.

While it would be ideal to evaluate our page placement policy on a real CC-NUMA GPU/CPU system with a heterogeneous memory system, such systems are not available today. Mobile systems containing both ARM CPU cores and NVIDIA GPU cores exist today in products such as the NVIDIA Shield Portable, but use a single LPDDR3 memory system. Desktop and HPC systems today have heterogeneous memory attached to CPUs and discrete GPUs but these processors are not connected through a cache coherent interconnect. They require explicit user management of memory if any data is to be copied from the host CPU memory to the GPU memory or vice versa over the PCIe interconnect. Pages can not be directly placed into GPU memory on allocation by the operating system. Without a suitable real system to experiment on, we turned to simulation to evaluate our

page placement improvements.

3.2.2 Methodology

3.2.2.1 Baseline system configuration

To evaluate page placement policies, we simulated a heterogeneous memory system attached to a GPU comprised of both bandwidth-optimized GDDR and cost/capacity-optimized DDR where the GDDR memory is attached directly to the GPU. No contemporary GPU system is available which supports cache-coherent access to heterogeneous memories. Commonly available PCIe-attached GPUs are constrained by interconnect bandwidth and lack of cache-coherence; while cache-coherent GPU systems, such as AMD's Kaveri, do not ship with heterogeneous memory. Our simulation environment is based on GPGPU-Sim [48] which has been validated against NVIDIA's Fermi class GPUs and is reported to match hardware performance with up to 98.3% accuracy [49]. We modified GPGPU-Sim to support a heterogeneous GDDR5-DDR4 memory system with the simulation parameters listed in Table 3.1. We model a baseline system with 200GB/s of GPU-attached memory bandwidth and 80GB/s of CPU-attached memory bandwidth, providing a bandwidth ratio of $2.5\times$ as shown in Table 3.1. We made several changes to the baseline GTX-480 model to bring our configuration in-line with the resources available in more modern GPUs, including a larger number of MSHRs and higher clock frequency.

3.2.2.2 Simulator modifications for policy evaluation

As noted in Section 2.1, attaching the capacity-optimized memory directly to the GPU is functionally equivalent to remote CPU attached memory, but with different latency parameters. To simulate an additional interconnect hop to remote CPU-attached memory, we model a fixed, pessimistic, additional 100 cycle latency to access the DDR4 memory from the GPU. This overhead is derived from the single additional interconnect hop latency found in SMP CPU-only designs such as the Intel XEON [37].

Our heterogeneous memory model contains the same number of MSHRs per memory channel as the baseline memory configuration. The number of MSHRs in the baseline configuration is sufficient to effectively hide the additional interconnect latency to the DDR memory in Figure 2.1b. Should MSHR quantity become an issue when supporting two level memories, previous work has shown that several techniques can efficiently increase MSHRs with only modest cost [50, 51].

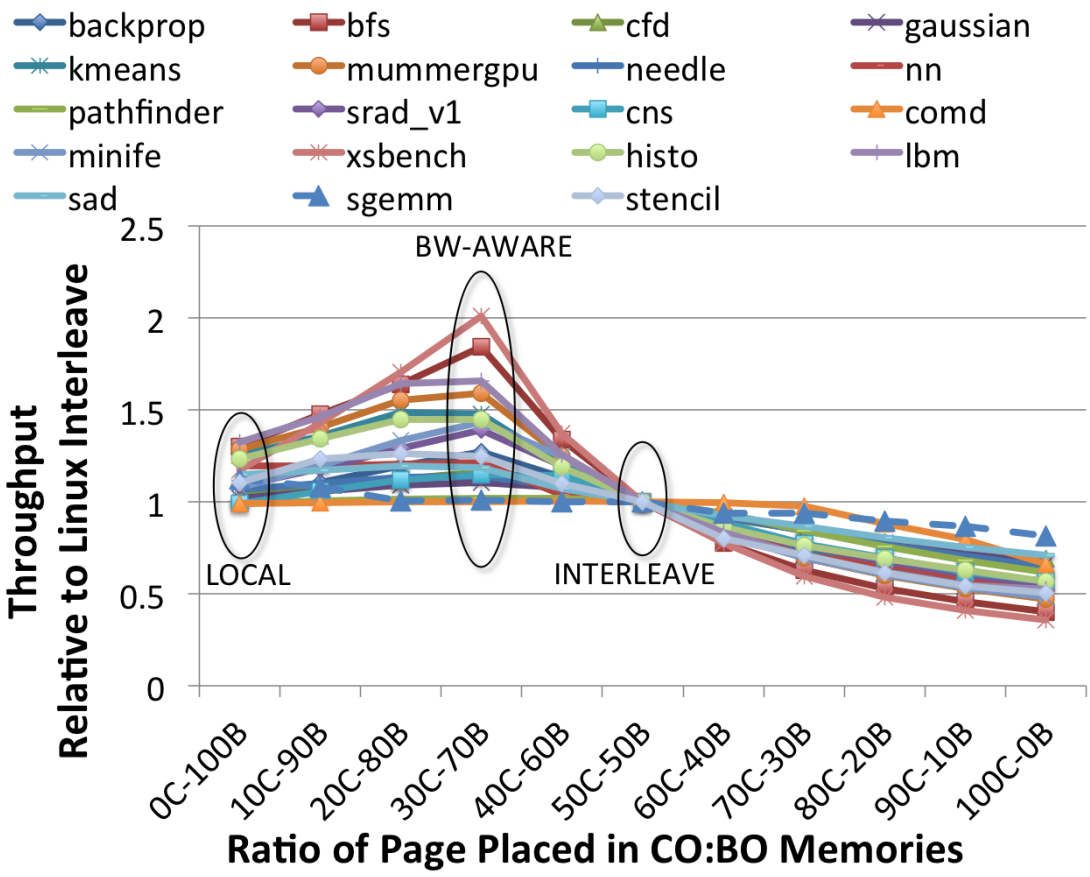


Figure 3.2: GPU workload performance with different page placement policies. xC-yB policy represents $x : y$ data transfer ratio from CO and BO memory respectively.

Implementing a BW-AWARE placement policy requires adding another mode (`MPOL_BWAWARE`) to the `set_mempolicy()` system call in Linux. When a process uses this mode, the Linux kernel will allocate pages from the two memory zones in the ratio of their bandwidths. These bandwidth ratios may be obtained from future ACPI resources or dynamically determined by the GPU runtime at execution time.

3.2.2.3 Benchmarks

With a focus on memory system performance, we evaluate GPU workloads which are sensitive to memory bandwidth or latency from three benchmark suites: Rodinia [27], Parboil [28] and recent HPC [29, 30, 31, 32] workloads; those which are compute-bound see little change in performance due to changes made to the memory system. For the remainder of the evaluation in this chapter we show results for 19 benchmarks, 17 of which are sensitive to memory bandwidth while also providing `comd` and `sgemm` results to represent applications which are memory insensitive and latency sensitive respectively.

3.2.3 BW-AWARE Performance: experimental results

We define our BW-AWARE page placement policy xC-yB, where x and y denote the percentage of pages placed in a given memory technology, C stands for capacity-optimized memory and B stands for bandwidth-optimized memory. By definition $x + y = 100$. For our baseline system with 200GB/sec bandwidth-optimized memory and 80GB/sec of capacity-optimized memory the aggregate system bandwidth is 280GB/sec. In this notation, our BW-AWARE policy will then be $x = 80/280 = 28\%$ and $y = 200/280 = 72\%$, represented as $28C$-$72B$. However, for simplicity we will round this to $30C$-$70B$ for use as the placement policy. For processes running on the GPU, the LOCAL policy would be represented as $0C$-$100B$; $50C$-$50B$ corresponds to the bandwidth spreading Linux INTERLEAVE policy.

To achieve the target $30C$-$70B$ bandwidth ratio, we implemented BW-AWARE placement as follows. On any new physical page allocation, a random number in the range $[0, 99]$ is generated. If this number is ≥ 30, the page is allocated from the bandwidth-optimized memory; otherwise it is allocated in the capacity-optimized memory. A LOCAL allocation policy can avoid the comparison if it detects either B or C has the value zero. While this implementation does not exactly follow the BW-AWARE placement ratio due to the use of random numbers, in practice this simple policy converges quickly towards the BW-AWARE ratio. This approach also requires no history of previous placements nor makes any assumptions about the frequency of access to pages, minimizing the overhead for making placement decisions which are on the software fast-path for memory allocation.

Figure 3.2 shows the application performance as we vary the ratio of pages placed in each type of memory from 100% BO to 100% CO. For all bandwidth-sensitive applications, the maximum performance is achieved when using the correct BW-AWARE $30C$-$70B$ placement ratio. We find that, on average, a BW-AWARE policy performs 18% better than the Linux LOCAL policy and 35% better than the Linux INTERLEAVE policy. However, for latency sensitive applications, such as `sgemm`, the BW-AWARE policy may perform worse than a LOCAL placement policy due to an increased number of accesses to higher latency remote CO memory. The BW-AWARE placement policy suffers a worse case performance degradation of 12% over the LOCAL placement policy in this scenario.

Because the current Linux INTERLEAVE policy is identical to BW-AWARE for a bandwidth-symmetric DRAM $50C$-$50B$ memory technology pairing, we believe a BW-AWARE placement policy could simply replace the current Linux INTERLEAVE policy without having significant negative side affects on existing CPU or GPU workloads. Because maximizing

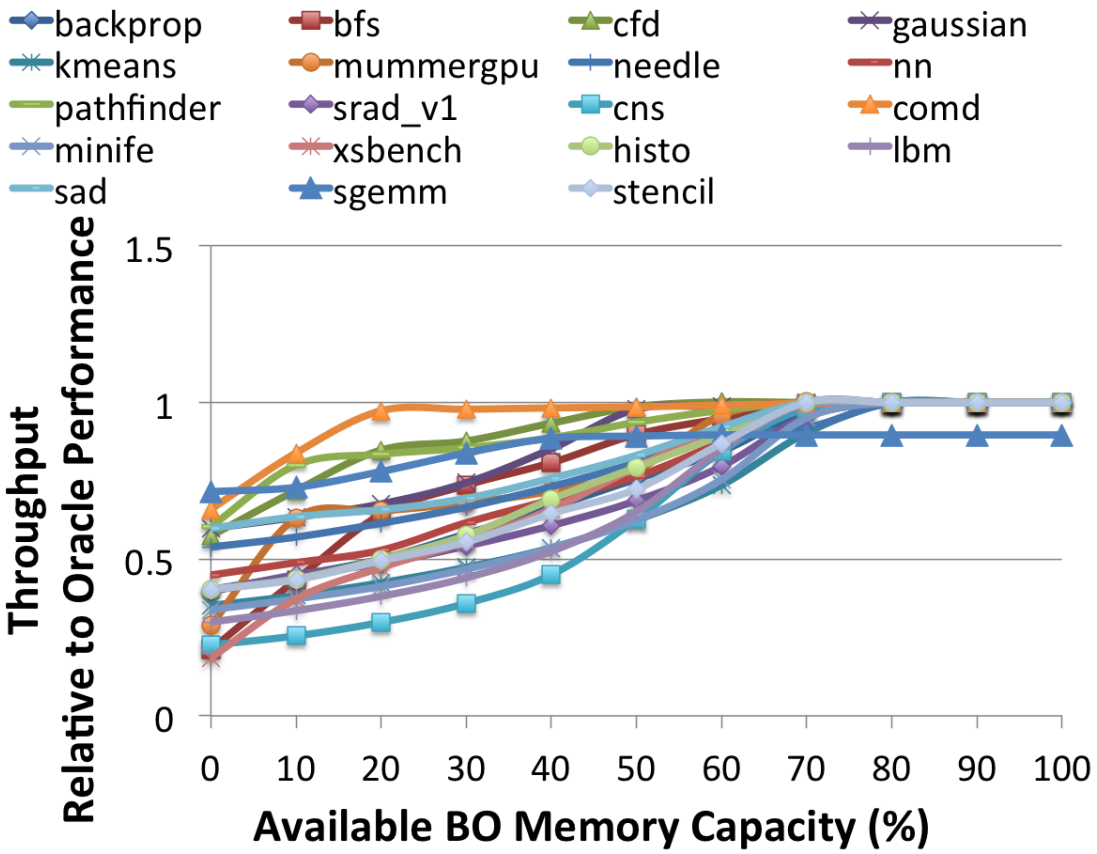


Figure 3.3: Performance of BW-AWARE placement as application footprint exceeds available high-bandwidth memory capacity.

bandwidth is more important than minimizing latency for GPU applications, BW-AWARE placement may be a good candidate to become the default placement policy for GPU-based applications.

3.2.3.1 Effective Improvement in Problem Sizing

Figure 5.8 shows the application throughput as we reduce the capacity of our bandwidth-optimized memory pool as a fraction of the total application footprint. BW-AWARE placement is able to achieve near peak performance even when only 70% of the application footprint fits within the BO memory because BW-AWARE placement places only 70% of pages in BO memory, with the other 30% is placed in the less expensive capacity-optimized memory. Thus, GPU programmers who today tune their application footprint to fit entirely in the GPU-attached BO memory could gain an extra 30% effective memory capacity by exploiting the CPU-attached CO memory with a BW-AWARE placement policy. However, as the bandwidth-optimized memory capacity drops to less than 70% of appli-

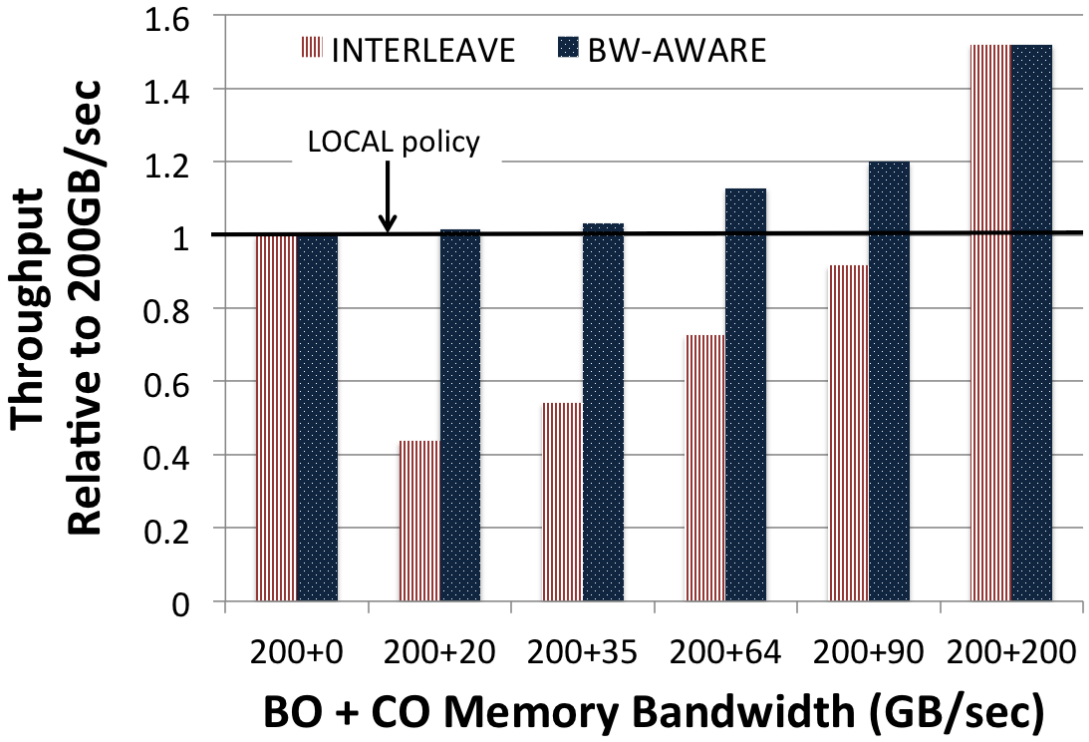


Figure 3.4: Performance comparison between BW-AWARE, INTERLEAVE, and LOCAL page placement policies while varying the memory bandwidth ratio.

cation footprint, performance begins to fall off. This effect is due to the ratio of bandwidth service from the two memory pools no longer matching the optimal ratio of $30C$-$70B$, with more data being serviced from the capacity optimized ratio than is ideal. Applications which are insensitive to memory bandwidth (shown as having little change in Figure 3.2), tend to maintain their performance at reduced capacity points (shown as having little change in Figure 5.8), because the average bandwidth reduction does not strongly affect their performance. Conversely, those applications with strong BW-performance scaling tend to see larger performance reduction as the average bandwidth available is reduced, due to capacity constraints forcing a disproportionate number of memory accesses to the lower bandwidth CO memory. The performance at 70% memory capacity does not exactly match 100% of ideal because the actual ratio of bandwidth in our system is $28C$-$72B$ not $30C$-$70B$.

3.2.3.2 Sensitivity to NUMA BW-Ratios

Heterogeneous memory systems are likely to come in a variety of configurations. For example, future mobile products may pair energy efficient and bandwidth-optimized Wide-

IO2 memory with cost-efficient and higher capacity LPDDR4 memory. Using the mobile bandwidths shown in Figure 3.1, this configuration provides an additional 31% in memory bandwidth to the GPU versus using the bandwidth-optimized memory alone. Similarly, HPC systems may contain GPUs with as many as 4 on-package bandwidth-optimized HBM stacks and high speed serial interfaces to bulk capacity cost/capacity-optimized DDR memory expanders providing just 8% additional memory bandwidth. While we have explored BW-AWARE placement in a desktop-like use case, BW-AWARE page placement can apply to all of these configurations.

Figure 3.4 shows the average performance of BW-AWARE, INTERLEAVE, and LOCAL placement policies as we vary the additional bandwidth available from the capacity-optimized memory from 0GB/s–200GB/s. As the bandwidth available from capacity-optimized memory increases, the LOCAL policy fails to take advantage of it by neglecting to allocate any pages in the capacity-optimized memory. The Linux INTERLEAVE policy, due to its fixed round-robin allocation, loses performance in many cases because it oversubscribes the capacity-optimized memory, resulting in less total bandwidth available to the application. On the other hand, BW-AWARE placement is able to exploit the bandwidth from the capacity-optimized memory regardless the amount of additional bandwidth available. Because BW-AWARE placement performs identically to INTERLEAVE for symmetric memory and outperforms it in all heterogeneous cases, we believe that BW-AWARE placement is a more robust default policy than INTERLEAVE when considering bandwidth-sensitive GPU workloads.

3.3 Understanding Application Memory Use

Section 3.2 showed that optimal BW-AWARE placement requires the majority of the application footprint to fit in the bandwidth-optimized memory to match the bandwidth service ratios of the memory pools. However, as shown in Figure 3.1, systems may have bandwidth-optimized memories that comprise less than 10% the total memory capacity, particularly those using on-package memories which are constrained by physical dimensions. If the application footprint grows beyond the bandwidth-optimized capacity needed for BW-AWARE placement, the operating system has no choice but to steer remaining page allocations into the capacity-optimized memory. Unfortunately, additional pages placed in the CO memory will skew the ratio of data transferred from each memory pool away from the optimal BW-AWARE ratio.

For example, in our simulated system if the bandwidth-optimized memory can hold just 10% of the total application memory footprint, then a BW-AWARE placement would

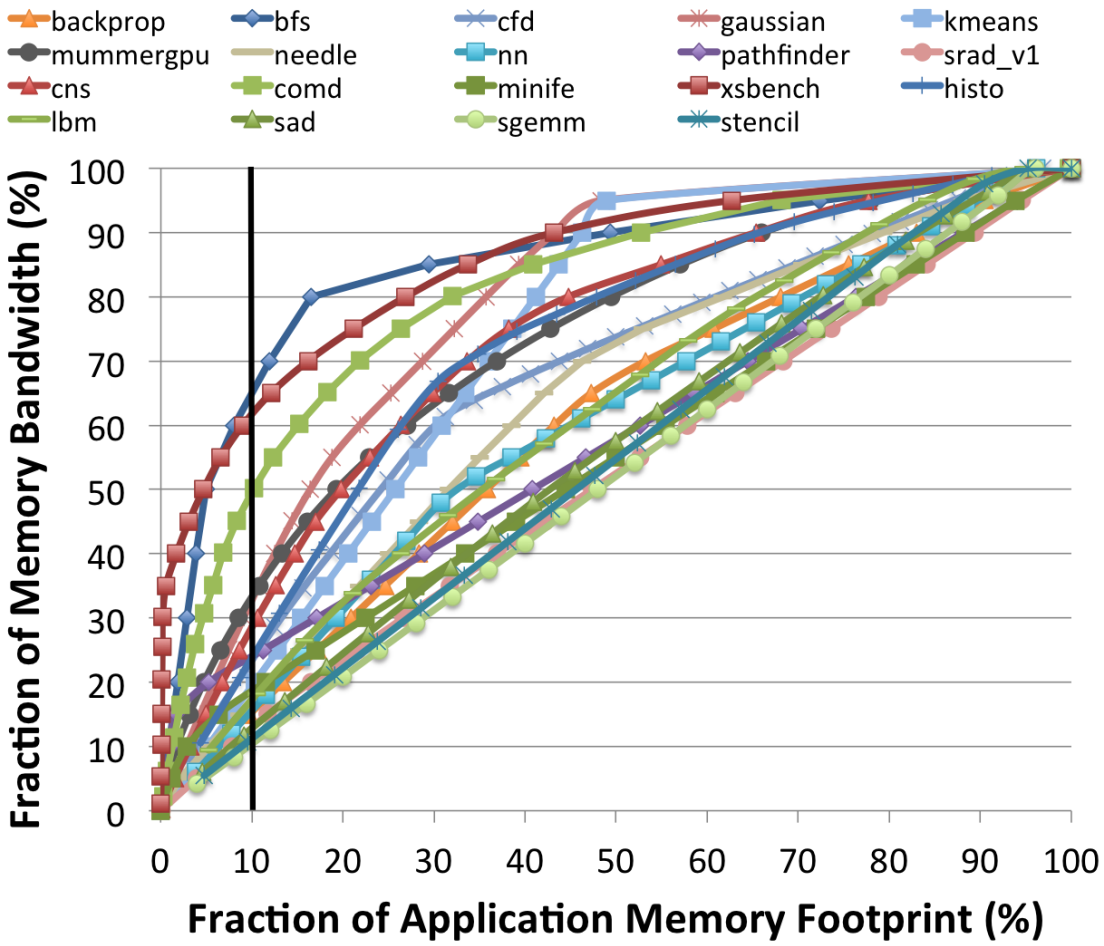


Figure 3.5: Data bandwidth cumulative distribution function with pages sorted from hot (most memory accesses) to cold (least memory accesses).

end up placing 10% pages in the BO memory; the remaining 90% pages must be spilled exclusively to the capacity-optimized memory. This ratio of $90C$-$10B$ is nearly the inverse of the performance-optimized ratio of $30C$-$70B$. To improve upon this capacity-induced placement problem, we recognize that not all pages have uniform access frequency, and we can selectively place hot pages in the BO memory and cold pages in the CO memory. In this work we define page *hotness* as the number of accesses to that page that are served from DRAM.

3.3.1 Visualizing Page Access Distribution

Figure 3.5 shows the cumulative distribution function (CDF) for memory bandwidth as a fraction of the total memory footprint for each of our workloads. The CDF was generated by counting accesses to each 4kB page, after being filtered by on-chip caches, and then

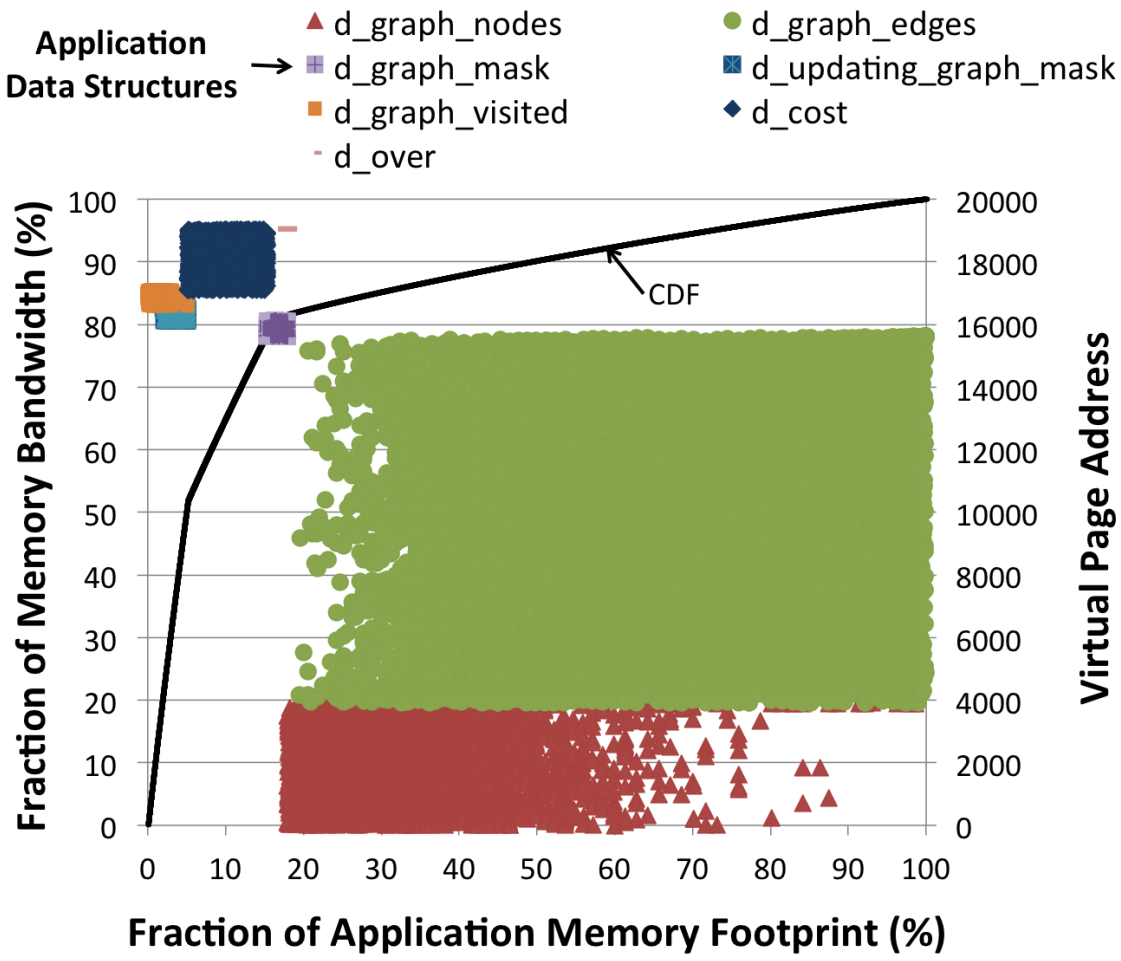


Figure 3.6: `bfs`: CDF of data footprint versus virtual address data layout.

sorting the pages from greatest to least number of accesses. Applications that have the same number of accesses to all pages have a linear CDF, whereas applications in which some pages are accessed more than others have a non-linear CDF skewed towards the left of the graph. For example, we see that for applications like `bfs` and `xsbench`, over 60% of the memory bandwidth stems from within only 10% of the application's allocated pages. Skewing the placement of these hot pages towards bandwidth-optimized memory will improve the performance of GPU workloads which are capacity constrained by increasing the traffic to the bandwidth-optimized memory. However, for applications which have linear CDFs, there is little headroom for improved page placement over BW-AWARE placement.

Figure 3.5 also shows that some workloads have sharp inflection points within the CDF, indicating that distinct ranges of physical addresses appear to be clustered as hot or cold. To determine if these inflection points could be correlated to specific memory allocations within the application, we plotted the virtual addresses that correspond to application

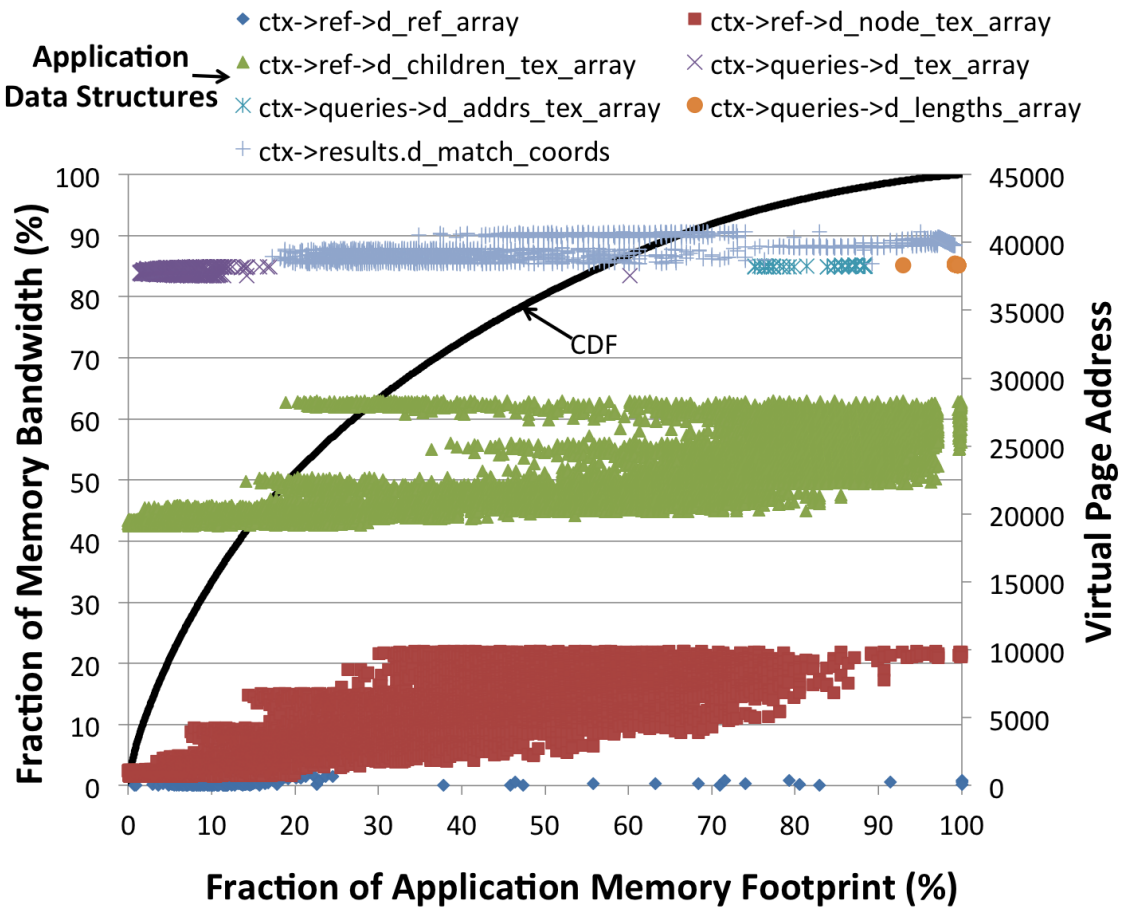


Figure 3.7: `mummergpu`: CDF of data footprint versus virtual address data layout.

pages in the CDF, and then reverse-mapped those address ranges to memory allocations for program-level data structures, with the results shown in Figure 3.6, 3.7, 3.8. The x-axis shows the fraction of pages allocated by the application, where pages are sorted from greatest to least number of accesses. The primary y-axis (shown figure left) represents the CDF of memory bandwidth among the pages allocated by the application (also shown in Figure 3.5). Each point on the secondary scatter plot (shown figure right) shows the virtual address of the corresponding page on the x-axis. The points (pages) are colored according to different data structures they were allocated from in the program source.

We analyze three example applications, `bfs`, `mummergpu`, and `needle` which show different memory access patterns. For `bfs`, we can see three data structures: `d_graph_visited`, `d_updating_graph_mask`, and `d_cost` consume $\approx 80\%$ of the total application bandwidth while accounting for $\approx 20\%$ of the memory footprint. However for `mummergpu`, the memory hotness does not seem to be strongly correlated to any specific application data structures. Several sub-structures have similar degrees of hotness and some vir-

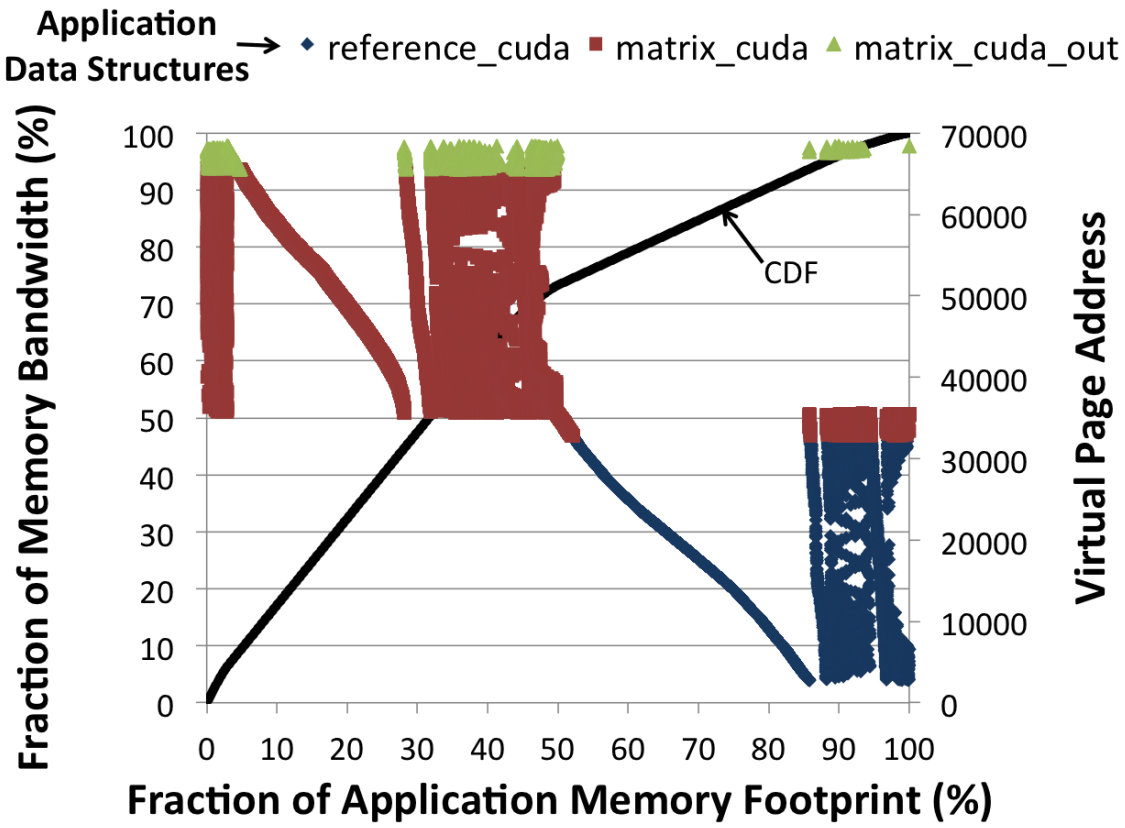


Figure 3.8: `needle`: CDF of data footprint versus virtual address data layout.

tual address ranges (10,000-19,000 and 29,000-37,000) are allocated but never never accessed. In `needle`, which has a fairly linear CDF, the degree of memory hotness actually varies within the same data structure. While we examined all applications with this analysis, we summarize our two key observations.

Individual pages can and often do have different degrees of hotness. Application agnostic page placement policies, including BW-AWARE placement, may leave performance on the table compared to a placement policy that is aware of page frequency distribution. Understanding the relative hotness of pages is critical to further optimizing page placement. If an application does not have a skewed CDF, then additional effort to characterize and exploit hotness differential will only introduce overhead without any possible benefit.

Workloads with skewed cumulative distribution functions often have sharp distinctions in page access frequency that map well to different application data structures. Rather than attempting to detect and segregate individual physical pages which may be hot or cold, application structure and memory allocation policy will likely provide good information about pages which will have similar degrees of hotness.

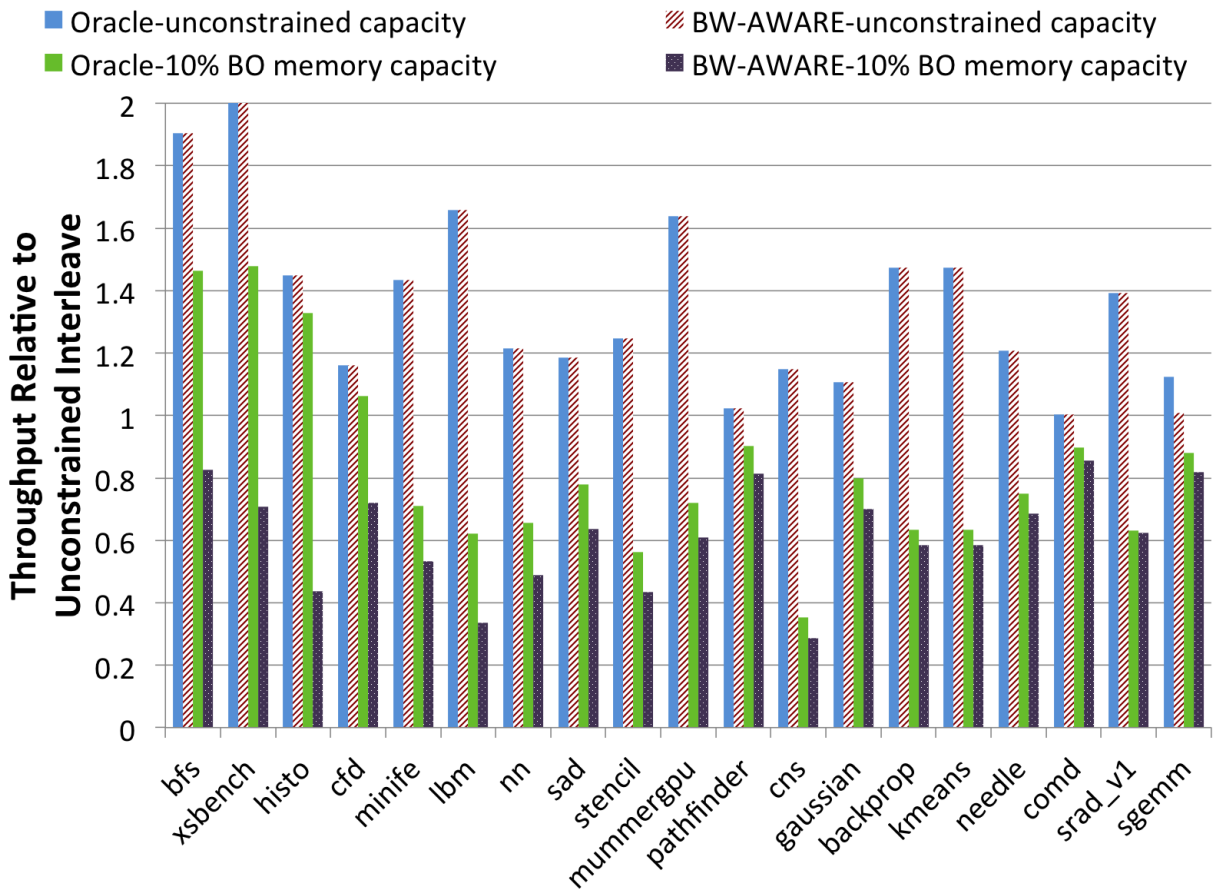


Figure 3.9: Oracle application performance of a constrained capacity system vs unconstrained capacity system

3.3.2 Oracle Page Placement

With knowledge that page accesses are highly differentiated for some applications, we implemented an oracle page placement policy to determine how much more application performance could be achieved compared to BW-AWARE placement in capacity constrained situations. Using perfect knowledge of page access frequency, an oracle page placement policy will allocate the hottest pages possible into the bandwidth-optimized memory until the target bandwidth ratio is satisfied, or the capacity of this memory is exhausted. We implemented this policy using two phase simulation. First, we obtained perfect knowledge of page access frequency. Then in a second simulation pass, we used this information to allocate pages to achieve the best possible data transfer ratio under a 10% capacity constraint where only 10% of the application memory footprint fits within the bandwidth-optimized memory.

Figure 3.9 compares the performance of the oracle and BW-AWARE placement policies in both the unconstrained and 10% capacity constrained configuration. Figure 3.9 confirms that BW-AWARE placement is near-optimal when applications are not capacity limited. This is because both BW-AWARE and oracle placement are both able to achieve the ideal bandwidth distribution, though the oracle policy is able to do this with a smaller memory footprint by placing fewer, hotter, pages in the BO memory. Under capacity constraints, however, the oracle policy can nearly double the performance of the BW-AWARE policy for applications with highly skewed CDFs and it outperforms BW-AWARE placement in all cases. This because the random page selection of BW-AWARE placement is not able to capture enough hot pages for placement in BO memory, before running out of BO memory capacity, to achieve the ideal bandwidth distribution. On average the oracle policy is able to achieve nearly 60% the application throughput of a system for which there is no capacity constraint. This additional performance, achievable through application awareness of memory access properties, motivates the need for further improvements in memory placement policy.

3.4 Application Aware Page Placement

Figure 3.6, 3.7, 3.8 visually depicts what may be obvious to performance-conscious programmers: *certain data structures are accessed more frequently than others.* For these programmers, the unbiased nature of BW-AWARE page placement is not desirable because all data structures are treated equally. This section describes compiler tool-flow and runtime modifications that enable programmers to intelligently steer specific allocations towards bandwidth- or capacity-optimized memories and achieve near-oracle page placement performance.

To correctly annotate a program to enable intelligent memory placement decisions across a range of systems, we need two pieces of information about a program: (1) the relative memory access hotness of the data structures, and (2) the size of the data structures allocated in the program. To understand the importance of these two factors, let us consider the following scenario. In a program there are two data structure allocations with hotness H1 and H2. If the bandwidth-optimized memory capacity is large enough for BW-AWARE placement to be used without running into capacity constraints, then BW-AWARE page placement should be used irrespective of the hotness of the data structures. To make this decision we must know the application runtime memory footprint. However, if the application is capacity-constrained, then ideally the memory allocation from the hotter data structure should be preferentially placed in the BO memory. In this case, we need

to know both the relative hotness and the size of the data structures to optimally place pages.

3.4.1 Profiling Data Structure Accesses in GPU Programs

While expert programmers may have deep understanding of their application characteristics, as machines become more complex and programs rely more on GPU accelerated libraries, programmers will have a harder time maintaining this intuition about program behavior. To augment programmer intuition about memory access behavior, we developed a new GPU profiler to provide information about program memory access behavior.

In this work we augmented `nvcc` and `ptxas`, NVIDIA's production compiler tools for applications written in CUDA [8] (NVIDIAs explicitly parallel programming model), to support data structure access profiling. When profiling is enabled, our compiler's code generator emits memory instrumentation code for all loads and stores that enables tracking of the relative access counts to virtually addressed data structures within the program. As with the GNU profiler `gprof` [52], the developer enables a special compiler flag that instruments an application to perform profiling. The developer then runs the instrumented application on a set of "representative" workloads, which aggregates and dumps a profile of the application.

When implementing this GPU memory profiling tool, one of the biggest challenges is that `nvcc` essentially generates two binaries: a host-side program, and a device-side program (that runs on the GPU). The profiler's instrumentation must track the execution of both binaries. On the host side, the profiler inserts code to track all instances and variants of `cudaMalloc`. The instrumentation associates the source code location of the `cudaMalloc` with the runtime virtual address range returned by it. The host side code is also instrumented to copy this mapping of line numbers and address ranges to the GPU before each kernel launch. The GPU-side code is instrumented by inserting code before each memory operation to check if the address falls within any of the ranges specified in the map.

For each address that falls within a valid range, a counter associated with the range is incremented, using atomic operations because the instrumentation code is highly multithreaded. At kernel completion, this updated map is returned to the host which launched the kernel to aggregate the statistics about virtual memory location usage. Our profiler generates informative data structure mapping plots, like those shown in

Figure 3.6, 3.7, 3.8, which application programmers can use to guide their understand-

ing of the relative access frequencies of their data structures, one of the two required pieces of information to perform intelligent near-optimal placement within an application.

3.4.2 Memory Placement APIs for GPUs

With a tool that provides programmers a profile of data structure hotness, they are armed with the information required to make page placement annotations within their application, but they are lacking a mechanism to make use of this information. To enable memory placement hints (which are not a functional requirement) for where data should be placed in a mixed BO-CO memory system, we also provide an alternate method for allocating memory. We introduce an additional argument to the `cudaMalloc` memory allocation functions that specifies in which domain the memory should be allocated (BO or CO) or to use BW-AWARE placement (BW). For example:

```
cudaMalloc(void **devPtr, size_t size, enum hint)
```

This hint is not machine specific and simply indicates if the CUDA memory allocator should make a best effort attempt to place memory within a BO or CO optimized memory using the underlying OS libNUMA functionality or fall back to the bandwidth-aware allocator. By providing an abstract hint, the CUDA runtime, rather than the programmer, becomes responsible for identifying and classifying the machine topology of memories as bandwidth or capacity optimized. While we have assumed bandwidth information is available in our proposed system bandwidth information table, programmatic discovery of memory zone bandwidth is also possible as a fall back mechanism [53]. In our implementation, memory hints are honored unless the memory pool is filled to capacity, in which case the allocator will fall back to the alternate domain. If no placement hint is provided, the CUDA runtime will fall back to using the application agnostic BW-AWARE placement for un-annotated memory allocations. When a hint is supplied, the `cudaMalloc` routine uses the `mbind` system call in Linux to perform placement of the data structure in the corresponding memory.

3.4.3 Program Annotation For Data Placement

Our toolkit now includes a tool for memory profile generation and a mechanism to specify abstract memory placement hints. While programmers may choose to use this information directly, optimizing for specific machines, making these hints performance portable across a range of machines is harder as proper placement depends on application footprint as well as the memory capacities of the machine. For performance portabil-

```
// n:input parameter
cudaMalloc(devPtr1, n*sizeof(int));
cudaMalloc(devPtr2, n*n);
cudaMalloc(devPtr3, 1000);
```

(a) Original code dependent allocations

```
// n: input parameter
// size[i]: Size of data structures
// hotness[i]: Hotness of data structures
size[0] = n*sizeof(int);
size[1] = n*n;
size[2] = 1000;
hotness[0] = 2;
hotness[1] = 3;
hotness[2] = 1;

// hint[i]: Computed data structure placement hints
hint[] = GetAllocation(size[], hotness[]);
cudaMalloc(devPtr1, size[0], hint[0]);
cudaMalloc(devPtr2, size[1], hint[1]);
cudaMalloc(devPtr3, size[2], hint[2]);
```

(b) Final code

Figure 3.10: Annotated pseudo-code to do page placement at runtime taking into account relative hotness of data structures and data structure sizes

ity, the hotness and allocation size information must be annotated in the program before any heap allocations occur. We enable annotation of this information as two arrays of values that are linearly correlated with the order of the memory allocations in the program. For example Figure 3.10 shows the process of hoisting the size allocations manually into the `size` array and hotness into the `hotness` array.

We provide a new runtime function `GetAllocation` that then uses these two pieces of information, along with the discovered machine bandwidth topology, to compute and provide a memory placement hint to each allocation. `GetAllocation` determines appropriate memory placement hints by computing the ideal (BO or CO) memory location by first calculating the cumulative footprint of all data structures and then calculating the total number of identified data structures from `[1:N]` that will fit within the bandwidth-optimized memory before it exhausts the BO capacity.

It is not critical that programmers provide annotations for all memory allocations, only large or performance critical ones. For applications that make heavy use of libraries or dynamic decisions about runtime allocations, it may not be possible to provide good hinting decisions because determining the size of data structures allocated within libraries calls is difficult, if not impossible, in many cases.. While this process may seem impractical to

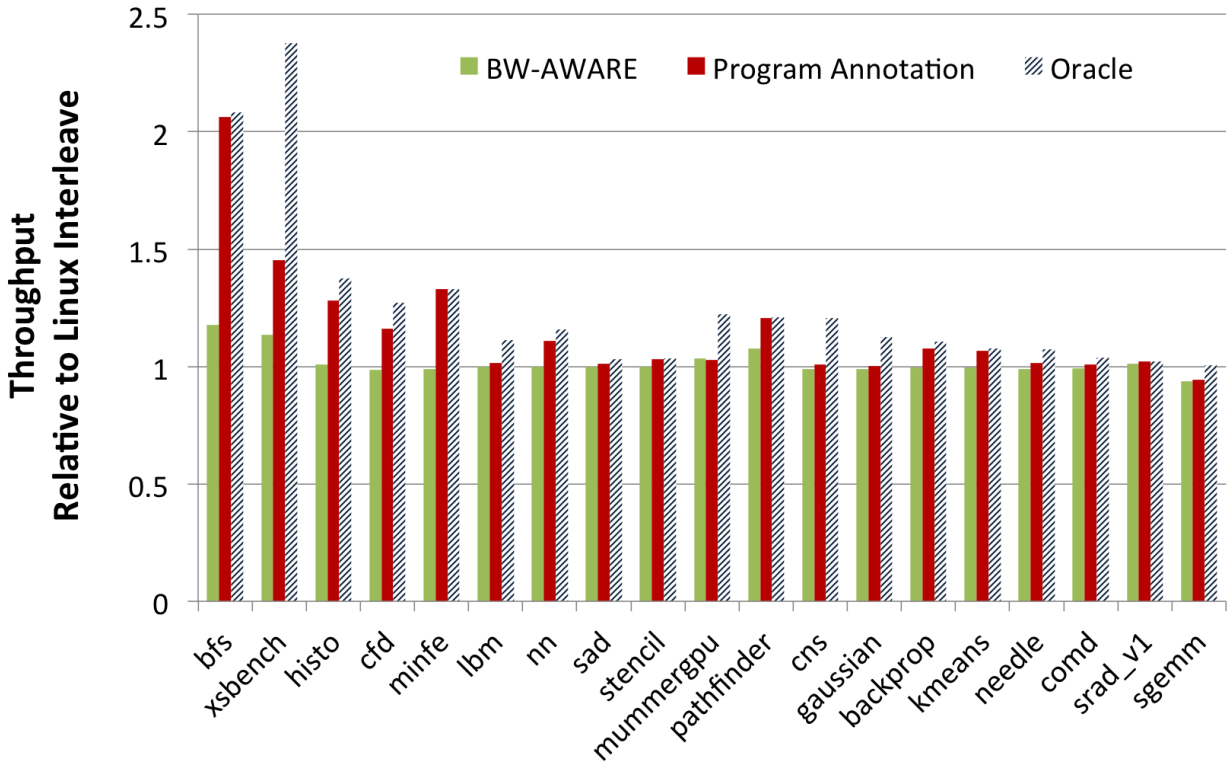


Figure 3.11: Profile-driven annotated page placement performance relative to INTERLEAVE, BW-AWARE and oracular policies under at 10% capacity constraint.

a traditional high level language programmer, examining a broad range of GPU compute workloads has shown that in almost all GPU-optimized programs, the memory allocation calls are already hoisted to the beginning of the GPU compute kernel. The CUDA C Best Practices Guide advises the programmer to minimize memory allocation and de-allocation in order to achieve the best performance [54].

3.4.4 Experimental Results

Figure 3.11 shows the results of using our feedback-based optimization compiler workflow and annotating our workloads using our new page placement interfaces. We found that on our capacity-limited machine, annotation-based placement outperforms the Linux INTERLEAVE policy performance by 19% and naive BW-AWARE 30C-70B placement by 14% on average. Combining program annotated placement hints and our runtime placement engine achieves 90% of oracular page placement on average. In all cases our program-annotated page placement algorithm outperforms BW-AWARE placement, making it a viable candidate for optimization beyond BW-AWARE placement if programmers choose to optimize for heterogeneous memory system properties.

One of the drawbacks to profile-driven optimization is that data dependent runtime characteristics may cause different behaviors than were seen during the profiling phase. While GPU applications in production will be run without code modification, the data set and parameters of the workload typically vary in both size and value from run-to-run. Figure 3.12 shows the sensitivity of our workload performance to data input set changes, where placement was trained on the first data-set but compared to the oracle placement for each individual dataset. We show results for the four example applications which saw the highest improvement of oracle placement over BW-AWARE. For `bfs`, we varied the number of nodes and average degree of the graph. For `xsbench`, we changed three parameters: number of nuclides, number of lookups, and number of gridpoints in the data set. For `minife`, we varied the dimensions of the finite element problem by changing the input matrix. Finally, for `mummergpu`, we changed the number of queries and length of queries across different input data sets.

Using the profiled information from only the training set, we observe that annotated placement performs 29% better than the baseline Linux INTERLEAVE policy, performs 16% better than our own BW-AWARE 30C-70B placement, and achieves 80% of the oracle placement performance. This result indicates that for GPU compute applications, feedback-driven optimization for page placement is not overly sensitive to application dataset or parameter variation, although pessimistic cases can surely be constructed.

3.4.5 Discussion

The places where annotation-based placement falls short primarily come from three sources. First, our application profiling relies on spatial locality of virtual addresses to determine page hotness. We have shown that this spatial locality holds true for many GPU applications, but this is not guaranteed to always be the case. Allocations within libraries or larger memory ranges the programmer chooses to logically sub-allocate within the program will not exhibit this property. The second shortcoming of our annotation-based approach is for applications which show high variance within a single data structure. For example, when using a hashtable where the application primarily accesses a small, dynamically determined portion of the total key-space, our static hotness profiling will fail to distinguish the hot and cold regions of this structure. Finally, although our runtime system abstracts the programming challenges of writing performance portable code for heterogeneous machines, it is still complex and puts a large onus on the programmer. Future work will be to learn from our current implementation and identify mechanisms to reduce the complexity we expose to the programmer while still making near-ideal page placement decisions.

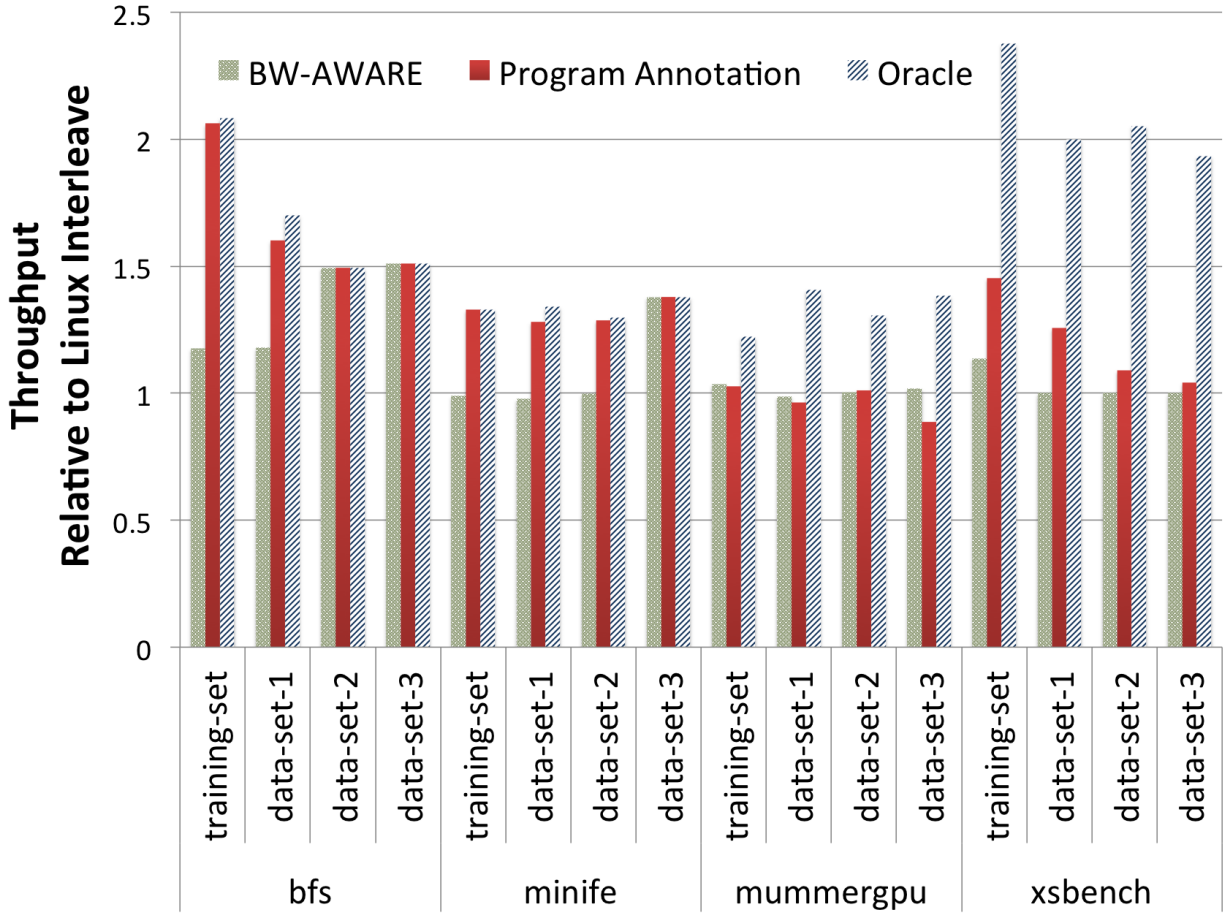


Figure 3.12: Annotated page placement effectiveness versus data sets variation after training phase under at 10% capacity constraint.

In this work we have focused on page placement for applications assuming a static placement of pages throughout the application runtime. We recognize that temporal phasing in applications may allow further performance improvement but have chosen to focus on initial page placement rather than page migration for two reasons. First, software-based page migration is a very expensive operation. Our measurements on the Linux 3.16-rc4 kernel indicate that it is not possible to migrate pages between NUMA memory zones at a rate faster than several GB/s and with several microseconds of latency between invalidation and first re-use. While GPUs can cover several hundred nanoseconds of memory latency, microsecond latencies encountered during migration will induce high overhead stalls within the compute pipeline. Second, online page migration occurs only after some initial placement decisions have been made. Focusing on online page migration before finding an optimized initial placement policy is putting the cart before of the horse. With improved default page placement for GPU workloads, the need for dynamic

page migration is reduced. Further work is needed to determine if there is significant value to justify the expense of online profiling and page-migration for GPUs beyond improved initial page allocation.

3.5 Conclusion

Current OS page placement policies are optimized for both homogeneous memory and latency sensitive systems. We propose a new BW-AWARE page placement policy that uses memory system information about heterogeneous memory system characteristics to place data appropriately, achieving 35% performance improvement on average over existing policies without requiring any application awareness. In future CC-NUMA systems, BW-AWARE placement improves the performance optimal capacity by better using all system resources. But some applications may wish to size their problems based on total capacity rather than performance. In such cases, we provide insight into how to optimize data placement by using the CDF of the application in combination with application annotations enabling intelligent runtime controlled page placement decisions. We propose a profile-driven application annotation scheme that enables improved placement without requiring any runtime page migration. While only the beginning of a fully automated optimization system for memory placement, we believe that the performance gap between the current best OS INTERLEAVE policy and the annotated performance (min 1%, avg 20%, max 2x) is enough that further work in this area is warranted as mobile, desktop, and HPC memory systems all move towards mixed CPU-GPU CC-NUMA heterogeneous memory systems.

CHAPTER IV

Unlocking Bandwidth for GPUs in CC-NUMA Systems

4.1 Introduction

GPUs are throughput oriented processors that spawn thousands of threads concurrently, demanding high memory bandwidth. To maximize the bandwidth utilization programmers copy over the data to high bandwidth memory like GDDR5 before launching GPU kernels to amortize the overhead of accessing memory over microsecond link latencies like PCIe. Hence, it is the responsibility of the programmer to identify data that will be accessed by the GPU and copy it over to the GPU-attached high bandwidth memory. NVIDIA's unified virtual memory [7] has relaxed this constraint to enhance GPU programmability by providing a software mechanism that performs on demand `memcpy` of the data as GPU accesses it. However, on demand data copying hurts GPU throughput. In this chapter we discuss techniques of performing programmer agnostic dynamic memory migration of performance critical data across CC-NUMA CPU-GPU system connected by a next generation interconnect technology to maximize bandwidth utilization, while not demanding the programmer to perform explicitly `memcpy(s)`. We specifically examine how to best balance accesses through cache-coherence and page migration.

4.2 Balancing Page Migration and Cache-Coherent Access

In the future, it is likely GPUs and CPUs will use a shared page table structure while maintaining local TLB caches. It remains to be seen if the GPU will be able to natively walk the operating system page tables to translate virtual to physical address information, or if GPUs will use an IOMMU-like hardware in front of the GPU's native TLB to perform such translations. In either case, the translation from virtual to physical addresses will be implicit, just as it is today for CPUs, and will no longer require trapping back to the CPU

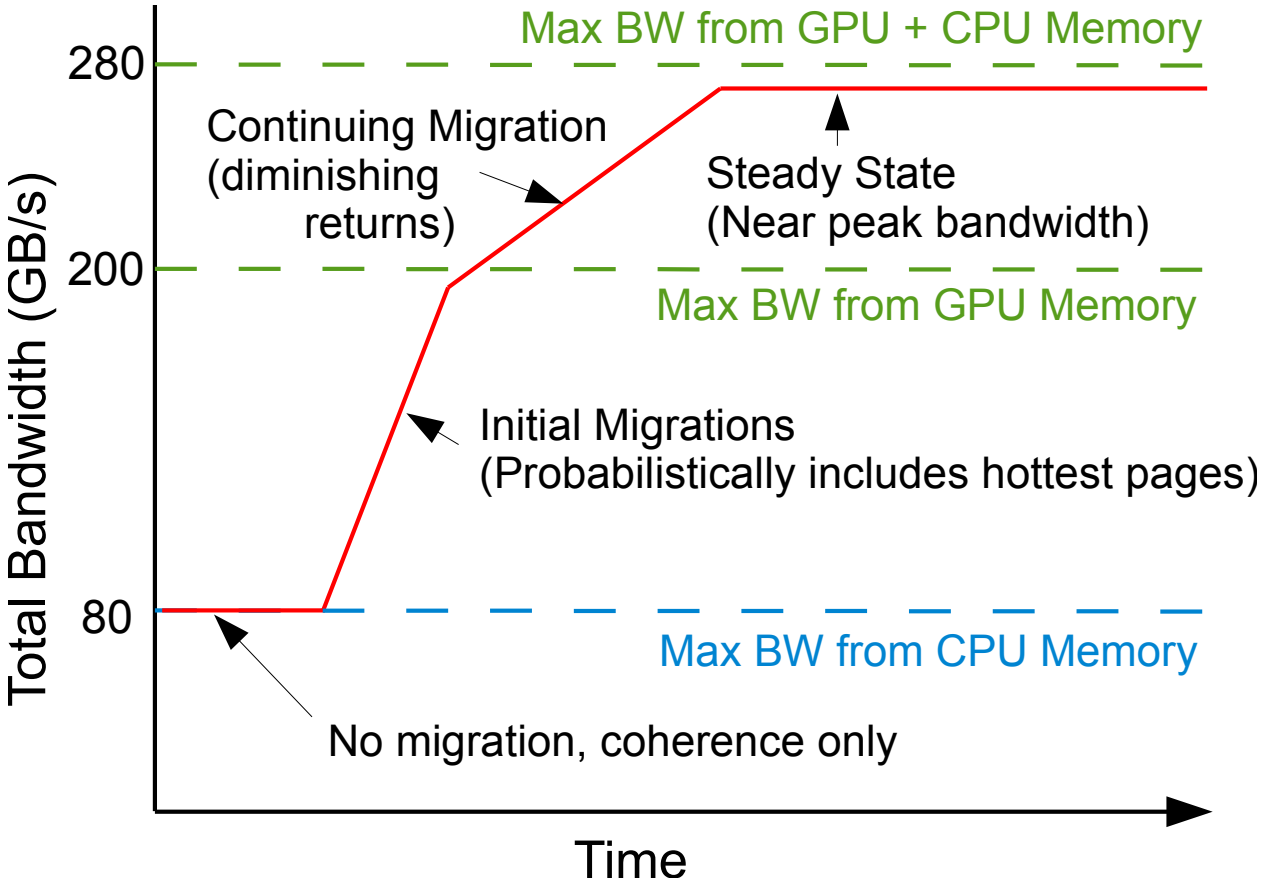


Figure 4.1: Opportunity cost of relying on cache coherence versus migrating pages near beginning of application run.

to translate or modify addresses. As a result, when page mappings must be modified, all CPUs—and now the GPU—must follow appropriate steps to safely invalidate their local TLB caches. While CPUs typically use a TLB per CPU-core, GPUs use a multi-level global page table across all compute pipelines. Therefore, when TLB shootdowns occur, the penalty will not stall just a single CPU pipeline, it is likely to stall the entire GPU. Whereas recent research has proposed intra-GPU sharer tracking [55] that could mitigate these stalls, this additional hardware is costly and typically unneeded for graphics applications and thus may not be adopted in practice.

Figure 4.1 provides a visual representation of the effect of balancing memory accesses from both DDR (CPU-attached) and GDDR (GPU-attached) memory. Initially, pages reside entirely in DDR memory. Without migration, the maximum bandwidth available to GPU accesses (via cache coherence to the DDR memory) will be limited by either the interconnect or DDR memory bandwidth. As pages are migrated from DDR to GDDR, the total bandwidth available to the GPU rises as pages can now be accessed concurrently

from both memories. Migrations that occur early in kernel execution will have the largest effect on improving total bandwidth, while later migrations (after a substantial fraction of GDDR memory bandwidth is already in use) have less effect. Performance is maximized when accesses are split across both channels in proportion to their peak bandwidth. Figure 4.1 shows the total bandwidth that is wasted if pages are not migrated eagerly, early in kernel execution. The key objective of the migration mechanism is to migrate the hottest pages as early as possible to quickly ramp up use of GDDR memory bandwidth. Nevertheless, migrating pages that are subsequently never accessed wastes bandwidth on both memory interfaces. In this section, we investigate alternative DDR-to-GDDR memory migration strategies. In particular, we contrast a simple, eager migration strategy against more selective strategies that try to target only hot pages.

4.2.1 Methodology

To evaluate page migration strategies, we model a GPU with a heterogeneous memory system comprising both GDDR and DDR memories. We discuss our baseline simulation framework in Chapter 3.2.2.1. Table 3.1 lists memory configuration of our simulation framework.

We model a software page migration mechanism in which migrations are performed by the CPU based on hints provided asynchronously by the GPU. The GPU tracks candidate migration addresses by maintaining a ring buffer of virtual addresses that miss in the GPU TLB. The runtime process on the CPU polls this ring buffer, converts the address to the page aligned base address and initiates migration using the standard Linux `move_pages` system call.

As in a CPU, the GPU TLB must be updated to reflect the virtual address changes that result from migrations. We assume a conventional x86-like TLB shootdown model where the entire GPU is treated like a single CPU using traditional inter-processor interrupt shootdown. In future systems, an IOMMU performing address translations on behalf of the GPU cores is likely to hide the specific implementation details of how it chooses to track which GPU pipelines must be stalled and flushed during any given TLB shootdown. For this work, we make a pessimistic assumption that, upon shootdown, all execution pipelines on the GPU must be flushed before the IOMMU handling the shootdown on behalf of the GPU can acknowledge the operation as complete. We model the time required to invalidate and refill the TLB entry on the GPU as a parameterized, fixed number, of cycles per page migration. In Section 4.2.2 we examine the effect of this invalidate/refill overhead on the performance of our migration policy, recognizing that the implementation

of TLB structures for GPUs is an active area of research [56, 57].

We model the memory traffic due to page migrations without any special prioritization within the memory controller and rely on the software runtime to rate-limit our migration bandwidth by issuing no more than 4 concurrent page migrations. We study our proposed designs using memory intensive workloads from Rodinia [27] and some other recent HPC applications [29, 30, 31, 32]. These benchmarks cover varied application domains, including graph-traversal, data-mining, kinematics, image processing, unstructured grid, fluid dynamics and Monte-Carlo transport mechanisms.

4.2.2 Results

To understand the appropriate balance of migrating pages early (as soon as first touch on the GPU) or later (when partial information about page hotness is known), we implemented a page migration policy in which pages become candidates for software controlled page migration only after they are touched N times by the GPU. Strictly migrating pages on-demand before servicing the memory requests will put page migration on the critical path for memory load latency. However, migrating a page after N references reduces the number of accesses that can be serviced from the GPU local memory, decreasing the potential impact of page migration. Once a page crosses the threshold for migration, we place it in an unbounded FIFO queue for migration, and allow the CUDA software runtime to migrate the pages by polling this FIFO and migrating pages as described in the previous sub-section.

To isolate the effect of choosing a threshold value from TLB shootdown costs, we optimistically assume a TLB shootdown and refill overhead of 0 cycles for the results shown in Figure 4.2. This figure shows application performance when migrating pages only after they have been touched N times, represented as threshold-N in the figure. The baseline performance of 1.0 reflects application performance if the GPU only accesses the CPU's DDR via hardware cache coherence and no page migrations to GDDR occur. Although we anticipated using a moderately high threshold (64–128) would generate the best performance (by achieving some level of differentiation between hot and cold data), the results in the figure indicate that, for the majority of the benchmarks, using the lowest threshold typically generates the best performance. Nevertheless, behavior and sensitivity to the threshold varies significantly across applications.

For the majority of our workloads, the best performance comes at a low migration threshold with performance degrading as the threshold increases. The peak performance is well above that achievable with only cache-coherent access to DDR memory, but it

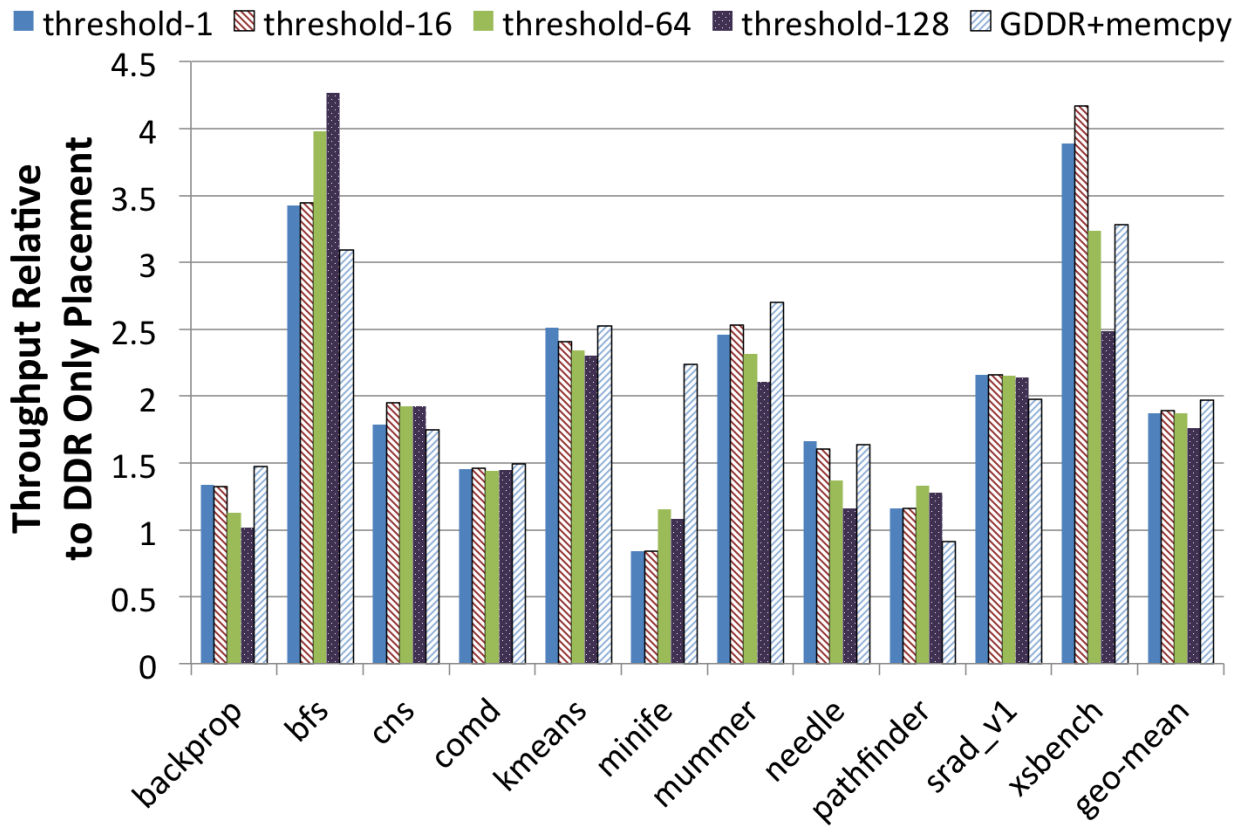


Figure 4.2: Performance of applications across varying migration thresholds, where threshold-N is the number of touches a given page must receive before being migrated from CPU-local to GPU-local memory.

rarely exceeds the the performance of the legacy `memcpy` programming practice. The `bfs` benchmark is a notable outlier, with higher migration thresholds improving performance by successfully differentiating hot and cold pages as candidates for migration. However, performance variation due to optimal threshold selection is much smaller than the substantial performance gain of using any migration policy. `Minife` is the second substantial outlier, with a low migration threshold decreasing performance below that of using CPU-only memory, while migration with higher thresholds provides only modest gains over cache-coherent access to DDR. Further analysis revealed that, for this workload, migration often occurs after the application has already performed the bulk of its accesses to a given page. In this situation, page migration merely introduces a bandwidth tax on the memory subsystem with little possibility for performance gain.

To implement a threshold-based migration system in practice requires tracking the number of times a given physical page has been touched. Such counting potentially requires tracking all possible physical memory locations that the GPU may access and

storing this side-band information either in on-chip SRAMs at the L2, memory controller, or within the DRAM itself. Additional coordination of this information may be required between the structures chosen to track this page-touch information. Conversely, a first touch policy (threshold-1) requires no tracking information and can be trivially implemented by migrating a page the first time the GPU translates an address for the page. Considering the performance differential seen across thresholds, we believe the overhead of implementing the necessary hardware counters to track all pages within a system to differentiate their access counts is not worth the improvement over a vastly simpler first-touch migration policy.

In Figure 4.2 we showed the performance improvement achievable when modeling the bandwidth cost of the page migration while ignoring the cost of the TLB shootdown, which will stall the entire GPU. At low migration thresholds, the total number of pages migrated is largest and thus application performance is most sensitive to the overhead of the TLB shootdown and refill. Figure 4.3 shows the sensitivity of application slowdown to the assumed cost of GPU TLB shootdowns for a range of client-side costs similar to those investigated by Villavieja et al. [55]. While the TLB invalidation cost in current GPUs is much higher, due to complex host CPU interactions, it is likely that TLB invalidation cost will drop substantially in the near future (due to IOMMU innovation) to a range competitive with contemporary CPUs (i.e., 100 clock cycles).

Because the GPU comprises many concurrently executing pipelines, the performance overhead of a TLB shootdown, which may require flushing all compute pipelines, is high; it may stalls thousands of execution lanes rather than a single CPU core. Figure 4.3 shows that moving from an idealized threshold of zero, to a realistic cost of one hundred reduces average performance by 16%. In some cases this overhead can negate the entire performance improvement achieved through page migration. To maximize the performance under page migration, our migration mechanism must optimize the trade-off between stalling the GPU on TLB shootdowns versus the improved memory efficiency of migrating pages to the GPU. One way to reduce this cost is to simply perform fewer page migrations, which can be achieved by increasing the migration threshold above the migrate-on-first-touch policy. Unfortunately, a higher migration threshold also decreases the potential benefits of migration. Instead, we will describe mechanisms that can reduce the number of required TLB invalidations simply through intelligent page selection while maintaining the first-touch migration threshold.

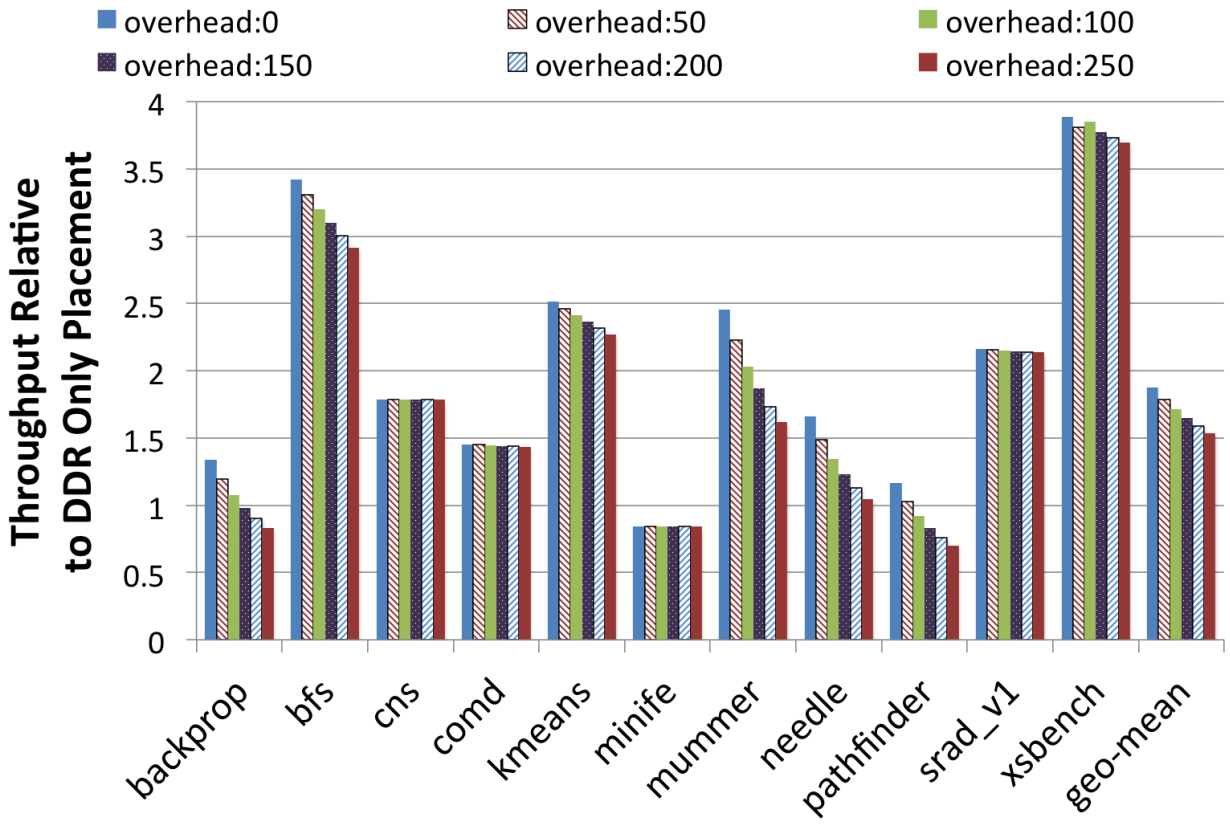


Figure 4.3: Performance overhead of GPU execution stall due to TLB shootdowns when using a first touch migration policy (threshold-1).

4.3 Range Expanding Migration Candidates

In the prior section, we demonstrated that aggressively migrating pages generally improves application performance by increasing the fraction of touches to a page serviced by higher-bandwidth (GPU-attached) GDDR versus (CPU-attached) DDR memory. This aggressive migration comes with high overheads in terms of TLB shootdowns and costly GPU pipeline stalls. One reason the legacy application directed `memcpy` approach works well is that it performs both aggressive up-front data transfer to GDDR and does not require TLB shootdowns and stalls. Unfortunately, this requirement for application-directed transfer is not well suited to unified globally addressable memory with dynamic allocation-based programming models. In this section, we discuss a prefetching technique that can help regain the performance benefits of bulk memory copying between private memories, without the associated programming restrictions.

Ideally, a page migration system prefetches pages into GDDR after they are allocated and populated in DDR, but before they are needed on the GPU. Studying the results of the threshold-based migration experiments, we observe that pages often are migrated

too late to have enough post-migration accesses to justify the cost of the migration. One way to improve the timeliness of migrations is via a prefetching scheme we call *range expansion*. Range expansion builds on the baseline single-page migration mechanism discussed previously. To implement basic range expansion, when the CUDA runtime is provided a virtual address to be migrated, the runtime also schedules an additional *N* pages in its (virtual address) neighborhood for migration. Pages are inserted into the migration queue in the order of furthest, from the triggered address, to the nearest, to provide the maximum prefetching affect based on spatial locality. We then vary this range expansion amount *N* from 0–128 and discuss the results in Section 4.3.2.

The motivation for migrating (virtually) contiguous pages can be seen in Figure 3.6, 4.4, 3.8. The figure shows virtual page addresses that are touched by the GPU for three applications in our benchmark set. The X-axis shows the fraction of the application footprint when sampled, after on-chip caches, at 4KB page granularity and sorted from most to fewest accesses. The primary Y-axis (shown figure left) shows the cumulative distribution function of memory bandwidth among the pages allocated by the application. Each point on the secondary scatter plot (shown figure right) shows the virtual address of the corresponding page on the x-axis. This data reveals that hot and cold pages are strongly clustered within the virtual address space. However, the physical addresses of these pages will be non-contiguous due to address interleaving performed by the memory controller. This clustering is key to range expansion because it suggests that if a page is identified for migration, then other neighboring pages in the virtual address space are likely to have a similar number of total touches. To exploit this property, range expansion migrates neighboring virtual addresses of a migration candidate *even if they have not yet been accessed on the GPU*. By migrating these pages before they are touched on the GPU, range expansion effectively prefetches pages to the higher bandwidth memory on the GPU, improving the timeliness and effectiveness of page migrations.

In the case that range expansion includes virtual addresses that are not valid, the software runtime simply allows the `move_pages` system call to fail. This scheme eliminates the need for additional runtime checking or data structure overhead beyond what is already done within the operating system as part of page table tracking. In some cases, a range expansion may extend beyond one logical data structure into another that is laid out contiguously in the virtual address space. While migrating these pages may be sub-optimal from a performance standpoint, there is no correctness issue with migrating these pages to GDDR. For HPC-style workloads with large, coarse-grained memory allocations, this problem happens rarely in practice.

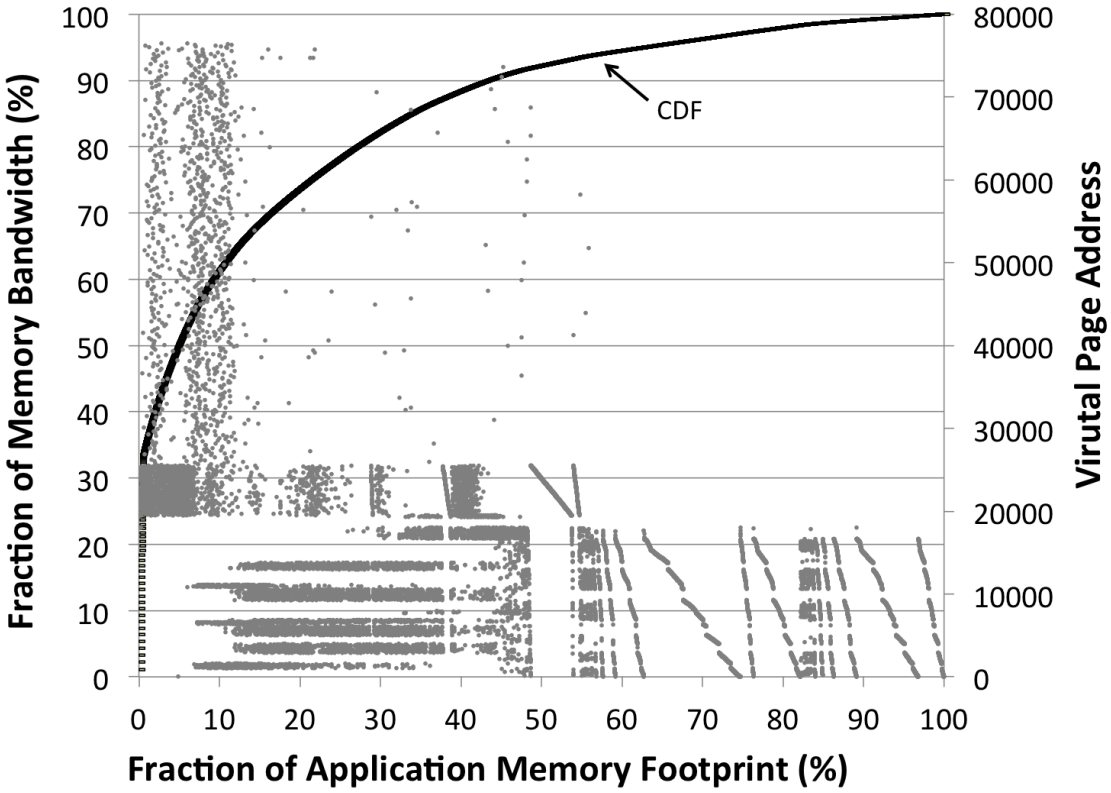


Figure 4.4: Cumulative distribution of memory bandwidth versus application footprint. Secondary axis shows virtual page address of pages touched when sorted by access frequency. (`xsbench`)

4.3.1 Avoiding TLB Shootdowns With Range Expansion

Figure 4.3 shows that TLB invalidations introduce significant overheads to DDR-to-GDDR migrations. Today, operating systems maintain a list of all processors that have referenced a page so that, upon modification of the page table, TLB shootdowns are only sent to those processor cores (or IOMMU units in the future) that may have a cached translation for this page. While this sharers list may contain false positives, because the mapping entry within a particular sharer may have since been evicted from their TLB, it guarantees that if no request has been made for the page table entry, that core will not receive a TLB shootdown request.

In our previous threshold-based experiments, pages are migrated after the GPU has touched them. This policy has the unfortunate side-effect that all page migrations will result in a TLB shootdown on the GPU. By using range expansion to identify candidates for migration that the GPU is likely to touch but has not yet touched, no TLB shootdown is required (as long as the page is in fact migrated before the first GPU touch). As a result,

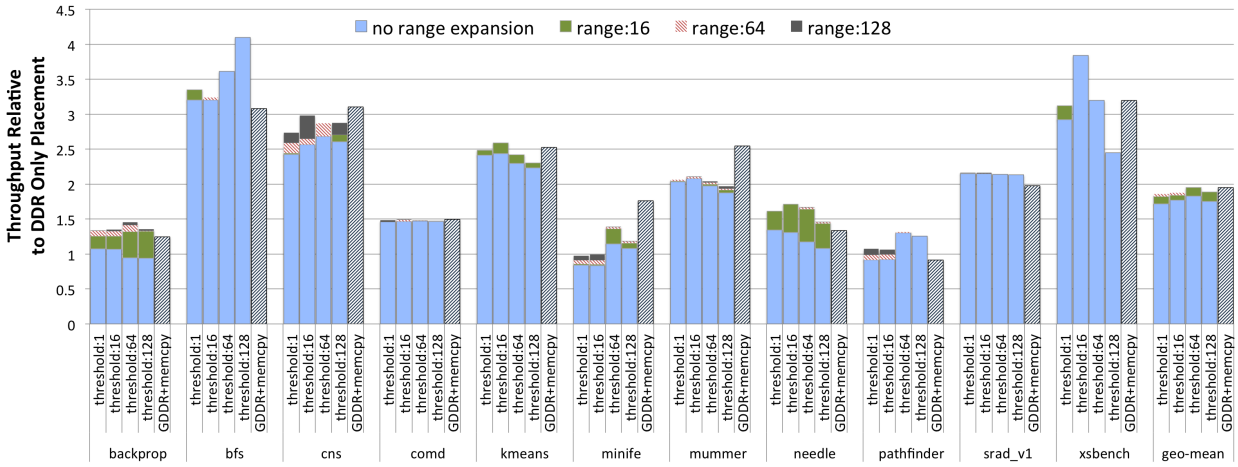


Figure 4.5: Effect of range expansion on workload performance when used in conjunction with threshold based migration.

range expansion provides the dual benefits of prefetching and reducing the number of costly TLB shootdowns.

4.3.2 Results

To evaluate the benefits of range expansion, we examine the effect that range expansion has when building on our prior threshold-based migration policies. We considered several thresholds from 1–128 accesses because, while the lowest threshold appears to have performed best in the absence of range expansion, it could be that using a higher threshold, thus identifying only the hottest pages, combined with aggressive range expansion would result in improved performance. We model a fixed TLB shootdown overhead of 100 cycles when performing these experiments, matching the baseline assumptions in the preceding section.

Figure 4.5 shows application performance as a stacked bar chart on top of the the baseline threshold-based page migration policy for various range expansion values. For the different range expansion values, a single migration trigger is expanded to the surrounding 16, 64, or 128 virtually addressed pages that fall within a single allocation (i.e., were allocated in the same call to `malloc`). The pages identified via range expansion are added to the page migration list in order of furthest, to nearest pages from the triggered virtual address. Pages that are farther from the (already accessed) trigger page are less likely to have been touched by the GPU yet and hence are least likely to be cached in the GPU's TLB. These pages therefore do not require expensive TLB shootdowns and pipeline stalls.

We see that range expansion allows us to outperform not only CC-NUMA access to DDR, but —in many cases— performance exceeds that of the legacy GDDR+`memcpy` implementation. These results indicate that aggressive prefetching, based on first touch access information, provides a balanced method of using both DDR and GDDR memory bandwidth. To understand the improvement from the reduction in TLB shootdowns, we report the fraction of page migrations that required no TLB shootdown in Table 4.1 (second column). Compared to threshold-based migrations without range expansion, where all migrations incur a TLB shootdown, range expansion eliminates 33.5% of TLB shootdowns on average and as many as 89% for some applications, drastically reducing the performance impact of these shootdowns.

Benchmark	Execution Overhead of TLB Shootdowns	% Migrations Without Shootdown	Execution Runtime Saved
backprop	29.1%	26%	7.6%
bfs	6.7%	12%	0.8%
cns	2.4%	20%	0.5%
comd	2.02%	89%	1.8%
kmeans	4.01%	79%	3.17%
minife	3.6%	36%	1.3%
mummer	21.15%	13%	2.75%
needle	24.9%	55%	13.7%
pathfinder	25.9%	10%	2.6%
srad_v1	0.5%	27%	0.14%
xsbench	2.1%	1%	0.02%
Average	11.13%	33.5%	3.72%

Table 4.1: Effectiveness of range prefetching at avoiding TLB shootdowns and runtime savings under a 100-cycle TLB shootdown overhead.

Figure 4.5 shows, for `bfs` and `xsbench`, that range expansion provides minimal benefit at thresholds > 1. In these benchmarks, the first touches to contiguous pages are clustered in time, because the algorithms are designed to use blocked access to the key data structures to enhance locality. Thus, the prefetching effect of range expansion is only visible when a page is migrated upon first touch to a neighboring page, by the second access to a page, all its neighbors have already been accessed at least once and there will be no savings from avoiding TLB shootdowns. On the other hand, in benchmarks such as `needle`, there is low temporal correlation among touches to neighboring pages. Even if a migration candidate is touched 64 or 128 times, some of its neighboring pages may not have been touched, and thus the prefetching effect of range expansion provides up to 42% performance improvement even at higher thresholds.

In the case of backprop, we can see that higher thresholds perform poorly compared to threshold 1. Thresholds above 64 are simply too high; most pages are not accessed this frequently and thus few pages are migrated, resulting in poor GDDR bandwidth utilization. Range expansion prefetches these low-touch pages to GDDR as well, recouping the performance losses of the higher threshold policies and making them perform similar to a first touch migration policy. For minife, previously discussed in subsection 4.2.2, the effect of prefetching via range expansion is to recoup some of the performance loss due to needless migrations. However, performance still falls short of the legacy memcpy approach, which in effect, achieves perfect prefetching. Overuse of range expansion hurts performance in some cases. Under the first touch migration policy (threshold-1), using range expansion 16, 64, and 128, the worst-case performance degradations are 2%, 3%, and 2.5% respectively. While not visible in the graph due to the stacked nature of Figure 4.5, they are included in the geometric mean calculations.

Overall, we observe that even with range expansion, higher-threshold policies do not significantly outperform the much simpler first-touch policy. With threshold 1, the average performance gain with range expansion of 128 is 1.85×. The best absolute performance is observed when using a threshold of 64 combined with a range expansion value of 64, providing 1.95× speedup. We believe that this additional ≈5% speedup over first touch migration with aggressive range expansion is not worth the implementation complexity of tracking and differentiating all pages in the system. In the next section, we discuss how to recoup some of this performance for benchmarks such as bfs and xsbench, which benefit most from using a higher threshold.

4.4 Bandwidth Balancing

In Section 4.2, we showed that using a static threshold-based page migration policy alone could not ideally balance migrating enough pages to maximize GDDR bandwidth utilization while selectively moving only the hottest data. In Section 4.3, we showed that informed page prefetching using a low threshold and range expansion to exploit locality within an application's virtual address space matches or exceeds the performance of a simple threshold-based policy. Combining low threshold migration with aggressive prefetching drastically reduces the number of TLB shootdowns at the GPU, reducing the performance overheads of page migration. These policies implemented together, however, will continue migrating pages indefinitely from their initial locations within DDR memory towards the GPU-attached GDDR memory.

As shown in Figure 4.2, however, rather than migrating all pages into the GPU memory,

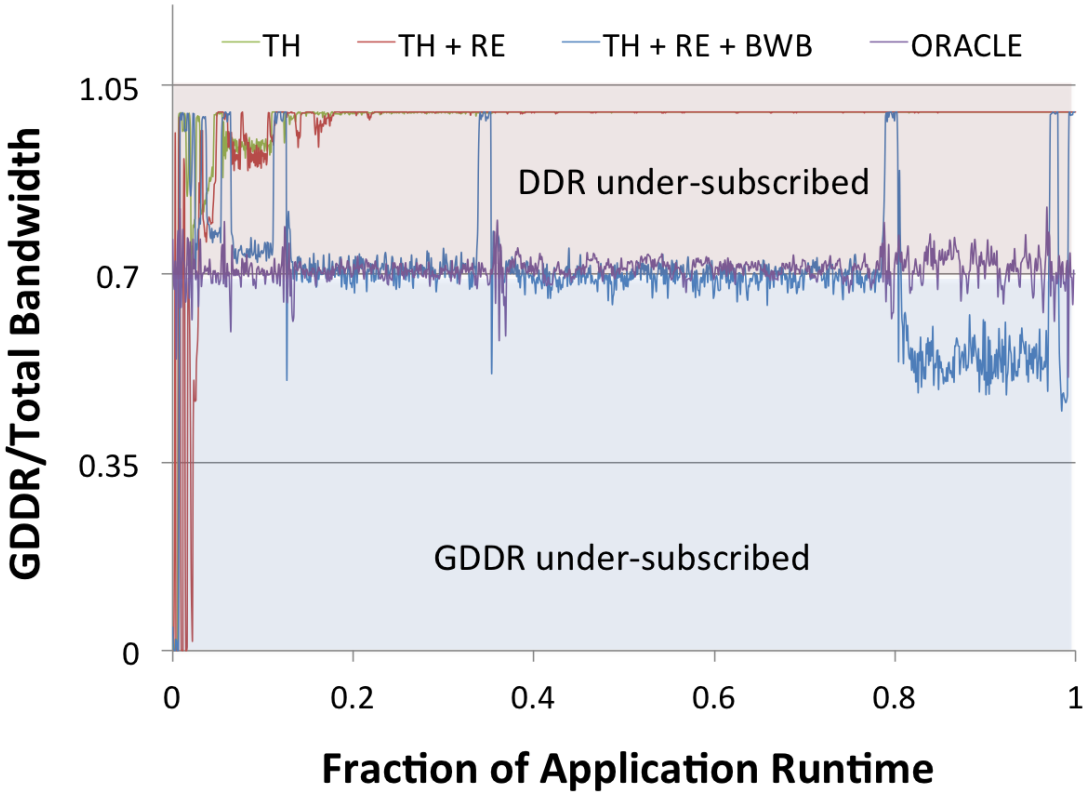


Figure 4.6: `bfs`: Fraction of total bandwidth serviced by GDDR during application runtime when when using thresholding alone (TH), then adding range expansion (TH+RE) and bandwidth aware migration (TH+RE+BWB).

optimal memory bandwidth utilization is achieved by migrating enough pages to GDDR to maximize its bandwidth while simultaneously exploiting the additional CPU DDR bandwidth via the hardware cache coherence mechanism. To prevent migrating too many pages to GDDR and over-shooting the optimal bandwidth target (70% of traffic to GDDR and 30% to DDR for our system configuration), we implement a migration rate control mechanism for bandwidth balancing. Bandwidth balancing, put simply, allows aggressive migration while the bandwidth ratio of GDDR to total memory bandwidth use is low, and rate limits (or eliminates) migration as this ratio approaches the system's optimal ratio. We implement a simple bandwidth balancing policy based on a sampled moving average of the application's bandwidth needs to each memory type. We assume that the ideal bandwidth ratio in the system can be known either via runtime discovery of the system bandwidth capabilities (using an application like stream [58]) or through ACPI bandwidth information tables, much like memory latency information can be discovered today.

Given the bandwidth capability of each interface, we can calculate the ideal fractional

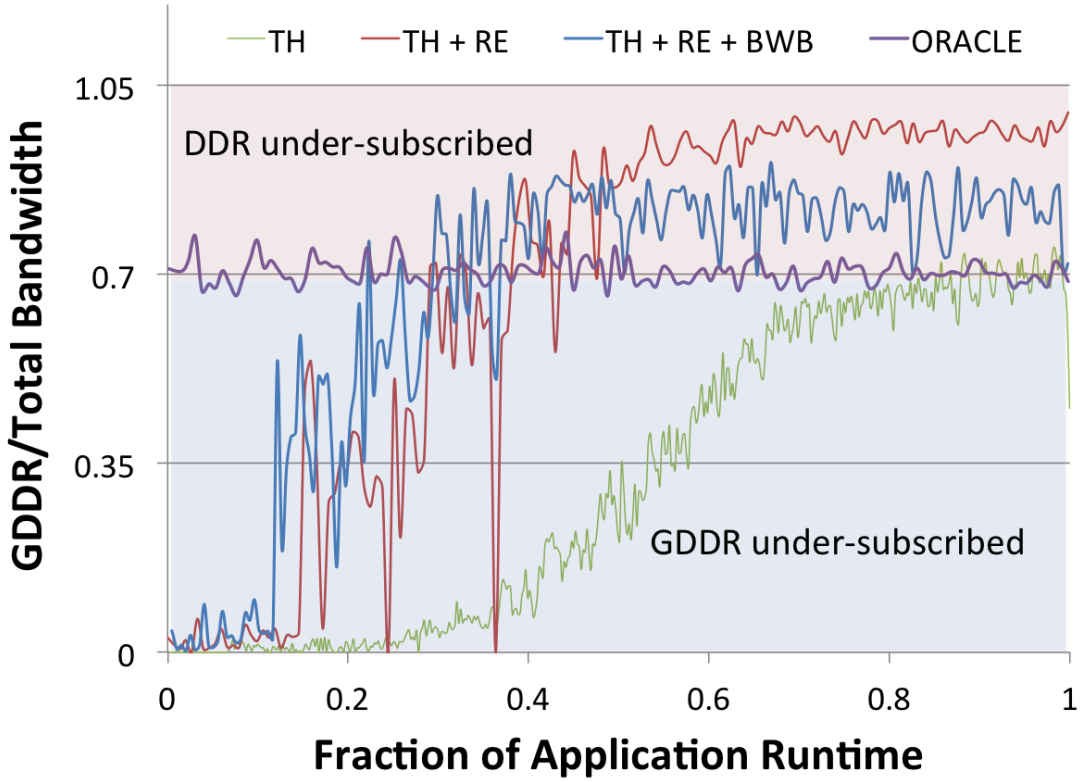


Figure 4.7: `xsbench`: Fraction of total bandwidth serviced by GDDR during application runtime when when using thresholding alone (TH), then adding range expansion (TH+RE) and bandwidth aware migration (TH+RE+BWB).

ratio, $GDDR/(DDR+GDDR)$, of traffic that should target GDDR using the methodology defined by Agarwal et al. [14]. For the configuration described in Table 3.1, this fraction is 71.4%. We currently ignore command overhead variance between the memory interfaces and assume that it is either the same for technologies in use or that the optimal bandwidth ratio discovered or presented by ACPI will have taken that into account. Using this target, our software page migration samples a bandwidth accumulator present for all memory channels every 10,000 GPU cycles and calculates the average bandwidth utilization of the GDDR and DDR in the system. If this utilization is below the ideal threshold minus 5% we continue migrating pages at full-rate. If the measured ratio approaches within 5% of the target we reduce the rate of page migrations by 1/2. If the measured ratio exceeds the target, we suspend further migrations.

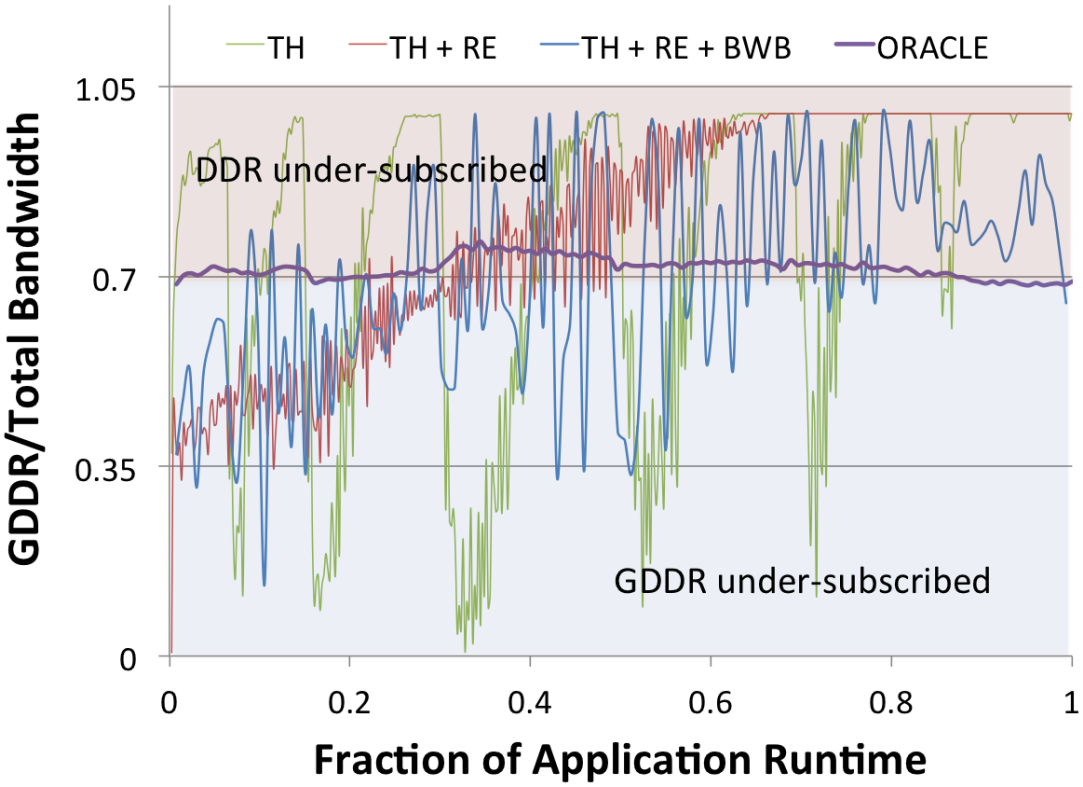


Figure 4.8: `needle`: Fraction of total bandwidth serviced by GDDR during application runtime when when using thresholding alone (TH), then adding range expansion (TH+RE) and bandwidth aware migration (TH+RE+BWB).

4.4.1 Results

For three example applications, Figure 4.6, 4.7, 4.8 shows the bandwidth utilization of the GDDR versus total bandwidth of the application sampled over time in 1% increments. The TH series provides a view of how migration using single page migration with a static threshold of one (first touch) performs, while $TH + RE$ shows the static threshold with the range expansion solution described in Section 4.3, and $TH + RE + BWB$ shows this policy with the addition of our bandwidth balancing algorithm. The oracle policy shows that if pages were optimally placed *a priori* before execution there would be some, but not more than 0.1% variance in the GDDR bandwidth utilization of these applications. It is also clear that bandwidth balancing prevents grossly overshooting the targeted bandwidth ratio, as would happen when using thresholds and range expansion alone.

We investigated various sampling periods shorter and longer than 10,000 cycles, but found that a moderately short window did not cause unwanted migration throttling during the initial migration phase but facilitated a quick adjustment of the migration policy once

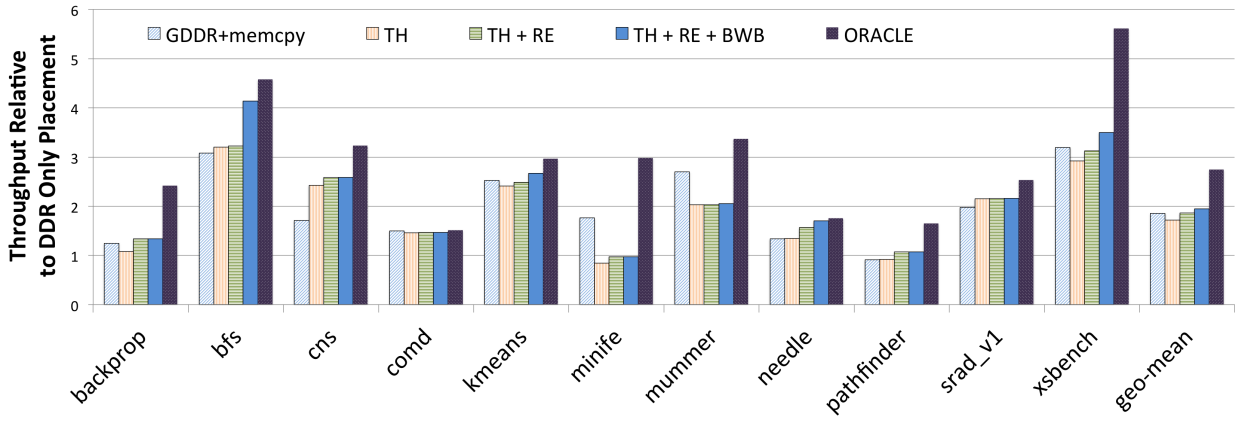


Figure 4.9: Application performance when using thresholding alone (TH), thresholding with range expansion (TH+RE), and thresholding combined with range expansion and bandwidth aware migration (TH+RE+BWB).

the target bandwidth balance was reached. If an application's bandwidth utilization subsequently dropped below the target, the short window again enabled rapid reaction to re-enable migration. While there is certainly room for further refinement (e.g., enabling reverse migration when the DDR memory becomes underutilized), our combined solution of threshold-based migration, prefetching via range expansion, and bandwidth balancing is able to capture the majority of the performance available by balancing page migration with CC-NUMA access. Figure 4.9 shows the results for our implemented solution across our benchmark suite. We see that, on average, we are able to not just improve upon CPU-only DDR by 1.95×, but also exceed the legacy up-front `memcpy`-based memory transfer paradigm by 6%, and achieve 28% of oracular page placement.

With our proposed migration policy in place, we seek to understand how it affects the overall bandwidth utilization. Figure 4.10 shows the fraction of total application bandwidth consumed, divided into four categories. The first, DDR Demand is the actual program bandwidth utilization that occurred via CC-NUMA access to the DDR. The second and third, DDR Migration and GDDR Migration, are the additional bandwidth overheads on both the DDR and GDDR that would not have occurred without page migration. This bandwidth is symmetric because for every read from DDR there is a corresponding write to the GDDR. Finally, GDDR Demand is the application bandwidth serviced from the GDDR. The two additional lines, DDR Oracle and GDDR Oracle, represent the ideal fractional bandwidth that could be serviced from each of our two memories.

We observe that applications which have the lowest GDDR Demand bandwidth see the least absolute performance improvement from page migration. For applications like `minife` and `pathfinder` the GDDR Migration bandwidth also dominates the GDDR De-

mand bandwidth utilized by the application. This supports our conclusion in subsection 4.2.2 that migrations may be occurring too late and our mechanisms are not prefetching the data necessary to make best use of GDDR bandwidth via page migration. For applications that do perform well with page migration, those that perform best tend to have a small amount of GDDR Migration bandwidth when compared to GDDR Demand bandwidth. For these applications, initial aggressive page migration quickly arrives at the optimal bandwidth balance where our bandwidth balancing policy then curtails further page migration, delivering good GDDR Demand bandwidth without large migration bandwidth overhead.

4.5 Conclusion

In this chapter we present a dynamic page migration policy that migrate pages to GPU-attached high bandwidth memory at application runtime without requiring any pro-

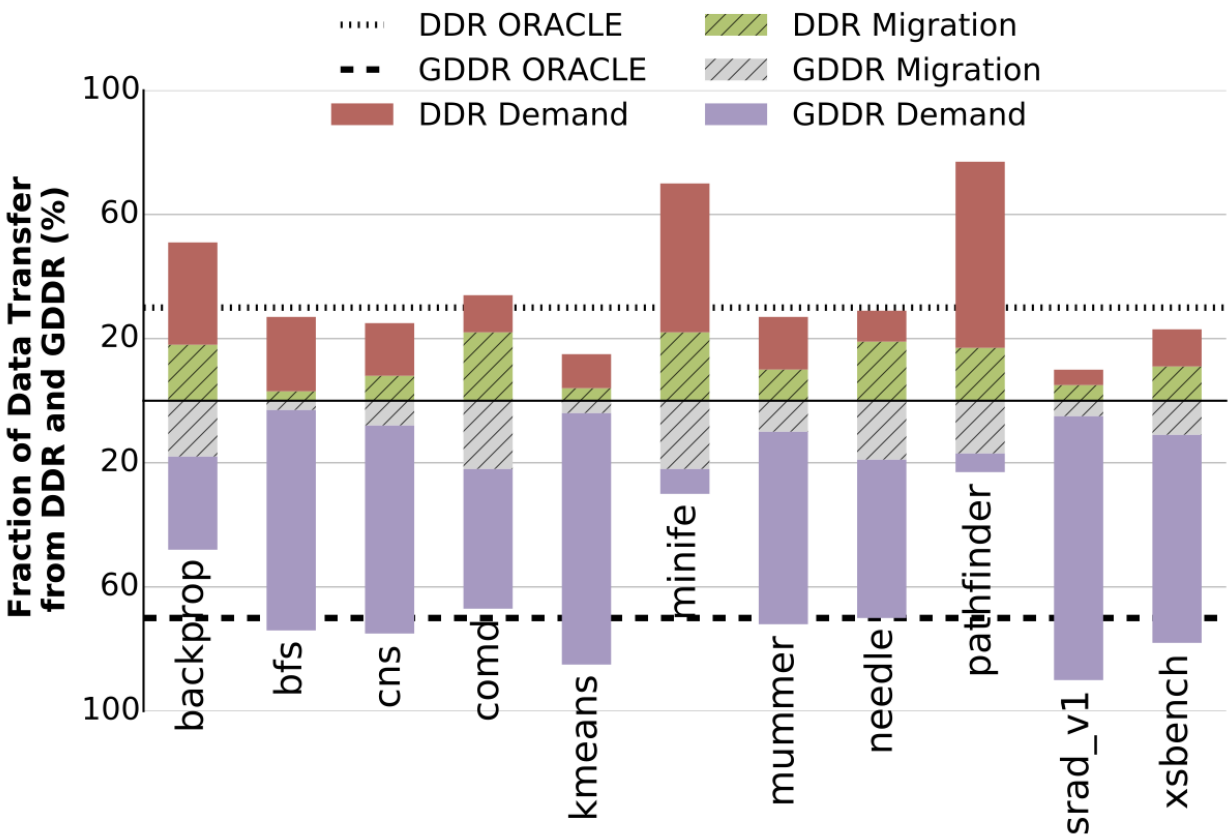


Figure 4.10: Distribution of memory bandwidth into demand data bandwidth and migration bandwidth

grammer involvement. We identify that demand-based migration alone is unlikely to be a viable solution due to both application variability and the need for aggressive prefetching of pages the GPU is likely to touch, but has not touched yet. The use of range expansion based on virtual address space locality, rather than physical page counters, provides a simple method for exposing application locality while eliminating the need for hardware counters. Our migration solution is able to outperform CC-NUMA access alone by 1.95×, legacy application `memcpy` data transfer by 6%, and come within 28% of oracular page placement.

CHAPTER V

Selective GPU Caches to Eliminate CPU–GPU Cache Coherence

5.1 Introduction

Technology trends indicate an increasing number of systems designed with CPUs, accelerators, and GPUs coupled via high-speed links. Such systems are likely to introduce unified shared CPU-GPU memory with shared page tables. In fact, some systems already feature such implementations [59]. Introducing globally visible shared memory improves programmer productivity by eliminating explicit copies and memory management overheads. Whereas this abstraction can be supported using only software page-level protection mechanisms [7, 15], hardware cache coherence can improve performance by allowing concurrent, fine-grained access to memory by both CPU and GPU. If the CPU and GPU have separate physical memories, page migration may also be used to optimize page placement for latency or bandwidth by using both near and far memory [60, 13, 61, 62].

Some CPU–GPU systems will be tightly integrated into a system on chip (SoC) making on-chip hardware coherence a natural fit, possibly even by sharing a portion of the on-chip cache hierarchy [15, 63, 21]. However, the largest GPU implementations consume nearly 8B transistors and have their own specialized memory systems [64]. Power and thermal constraints preclude single-die integration of such designs. Thus, many CPU–GPU systems are likely to have discrete CPUs and GPUs connected via dedicated off-chip interconnects like NVLINK (NVIDIA), CAPI (IBM), HT (AMD), and QPI (INTEL) or implemented as multi-chip modules [10, 65, 11, 12, 66]. The availability of these high speed off-chip interconnects has led both academic groups and vendors like NVIDIA to investigate how future GPUs may integrate into existing OS controlled unified shared memory regimes used by CPUs [56, 57, 14, 13].

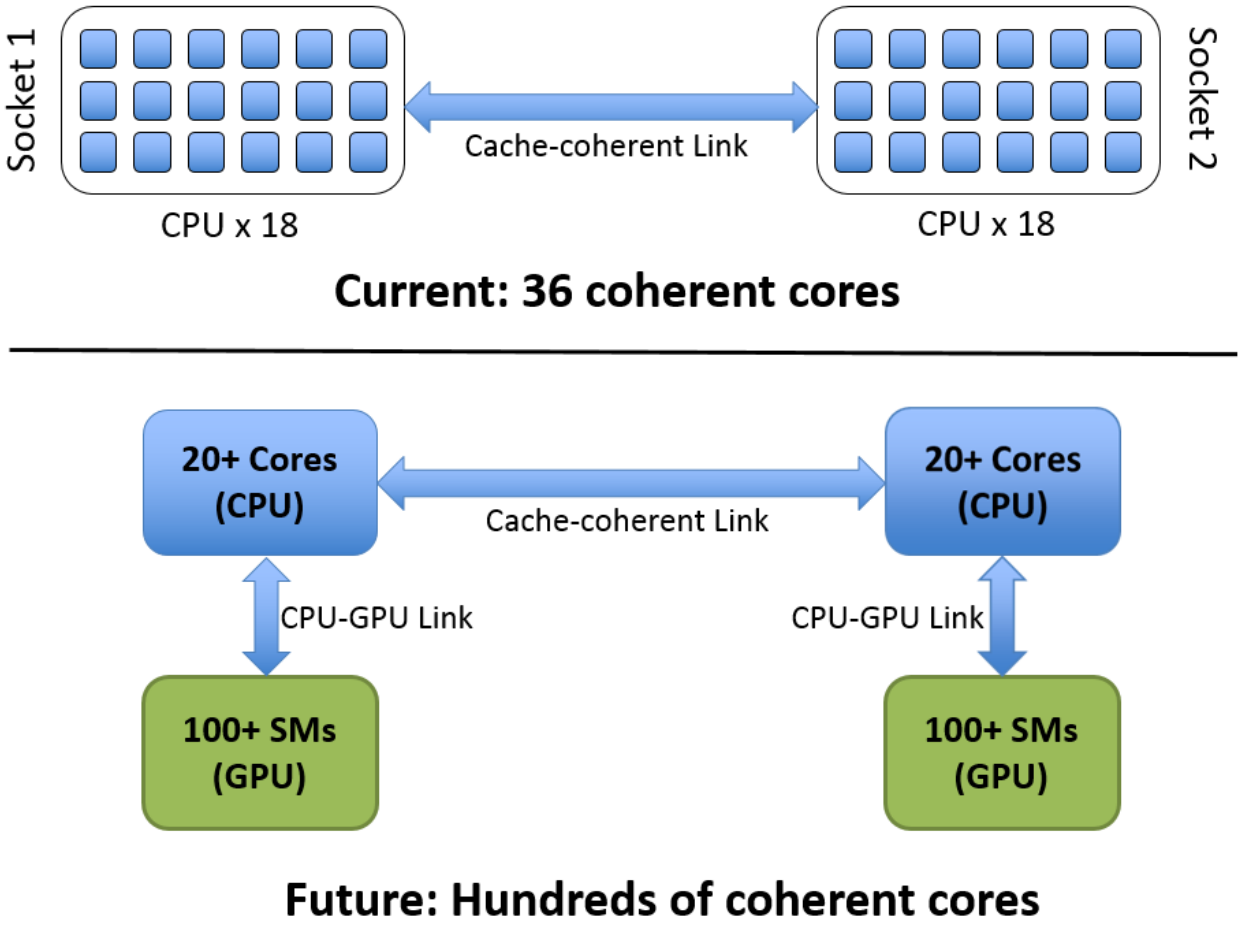


Figure 5.1: Number of coherent caches in future two socket CPU-only vs CPU–GPU systems.

Upcoming GPUs are expected to have hundreds of streaming multiprocessors (SMs) – as shown in Figure 6.1 – making the design of hardware cache coherence challenging. Coherence messages will have to be exchanged across the CPU-GPU interconnect requiring large states and interconnect bandwidth [19, 20]. In the past, NVIDIA has investigated extending hardware cache-coherence mechanisms to multi-chip CPU–GPU memory systems. In this chapter we explore the techniques to simplify the implementation of shared virtual address space in heterogeneous CPU-GPU systems by providing the programmers with the hardware cache coherence while still maintaining performance. We architect a GPU *selective caching* mechanism, wherein the GPU does not cache data that resides in CPU physical memory, nor does it cache data that resides in the GPU memory that is actively in-use by the CPU on-chip caches. This approach is orthogonal to the memory consistency model and leverages the latency tolerant nature of GPU architectures combined with upcoming low-latency and high-bandwidth interconnects to enable the benefits of shared memory. To evaluate the performance of such a GPU, we measure

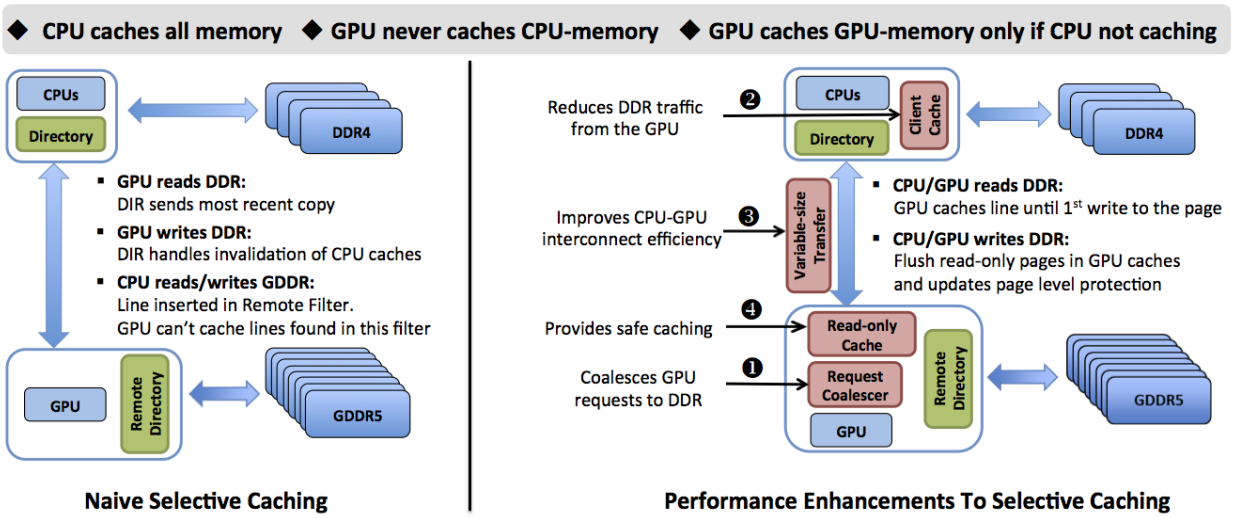


Figure 5.2: Overview of naive selective caching implementation and optional performance enhancements. Selective caching GPUs maintain memory coherence with the CPU while not requiring hardware cache coherence within the GPU domain.

ourselves against a theoretical hardware cache-coherent CPU–GPU system that, while high performance, is impractical to implement.

5.2 GPU Selective Caching

Historically, GPUs have not required hardware cache coherence because their programming model did not provide a coherent address space between threads running on separate SMs [67]. CPUs however, support hardware cache coherence because it is heavily relied upon by both system and application programmers to ensure correctness in multi-threaded programs and to provide simpler programming models for multi-core systems. Existing GPU programming models do not guarantee data correctness when CPU and GPU accesses interleave on the same memory location while the GPU is executing. One way to provide such guarantees is to enforce CPU-GPU hardware cache coherence, albeit with significant implementation complexity as previously discussed in Chapter 2.5.

Alternatively, if the GPU does not cache any data that is concurrently cached by the CPU, no hardware coherence messages need to be exchanged between the CPU and GPU, yet data correctness is still guaranteed. This approach also decouples the, now private, coherence protocol decisions in CPU and GPU partitions, facilitating multi-vendor system integration. We now discuss how CPU–GPU memory can provide this single shared memory abstraction without implementing hardware cache coherence. We then

propose several micro-architectural enhancements to enable selective caching to perform nearly as well as hardware cache coherence, while maintaining the programmability benefits of hardware cache coherence.

5.2.1 Naive Selective Caching

As shown in Figure 5.2, three simple principles enable the GPU to support a CPU-visible shared memory by implementing selective caching. First, the CPU is always allowed to cache any data in the system regardless of whether that data is physically located in the memory attached to the GPU or the CPU. Second, the GPU is never allowed to cache data that resides within the CPU memory. Finally, the GPU may cache data from its own local memory if and only if the CPU is not also caching a copy of this data.

When the CPU is known to be caching a line that is homed in GPU memory and the GPU requests this line, the request must be routed to the CPU where the requested data is serviced from the CPU cache, rather than the GPU memory. Similarly, if the GPU is caching a line that the CPU requests, then this line must be flushed from the GPU caches when the request is received by the GPU memory controller. By dis-allowing caching of memory in use by the CPU, the GPU cannot violate the CPU hardware coherence model.

The primary microarchitectural structure needed by the GPU to implement selective caching is the *remote directory*. The remote directory block shown in Figure 5.2 (in green) tracks approximately, but conservatively, the cache lines homed in GPU memory that are presently cached at the CPU. When the CPU requests a line from GPU memory, its cache block address is entered into the remote directory. If the address was not already present, the GPU probes and discards the line from all GPU caches, as in a conventional invalidation-based coherence protocol. Once a cache block is added to the GPU remote directory, it becomes un-cacheable within the GPU; future GPU accesses to the line will be serviced from the CPU cache.

To limit hardware cost, we implement the remote directory as a cuckoo filter (a space efficient version of a counting bloom filter) that never reports false negatives but may report false positives [68, 69]. Thus, the remote directory may erroneously, but conservatively, indicate that a line is cached at the CPU that has never been requested, but will accurately reference all lines that have actually been requested. False positives in the remote directory generate a spurious request to the CPU, which must respond with a negative acknowledgement (NACK) should the line not be present in the CPU cache. This request will then be serviced from the GPU memory system. Similarly, if the CPU has cached a line homed in GPU memory (causing a remote directory insertion) and has

since evicted it, the CPU may also NACK a GPU request, causing the request to return to the GPU memory for fulfillment.

Because entries are inserted but never pruned from our remote directory, we must track if the directory becomes full or reaches a pre-selected high-water mark. If it becomes full, our implementation forces the CPU to flush all cache lines homed in GPU memory and then resets the remote directory. This limited cache flush operation does not flush any lines homed in CPU memory, the vast majority of the system's memory capacity. In our design, the flush is performed by triggering a software daemon to call the Linux `cacheflush` trap.

The remote directory is sized to track CPU caching of up to 8MB of GPU memory, which when fully occupied requires just 64KB of on-chip storage to achieve a false positive rate of 3%. In the workloads we evaluate, the remote directory remains largely empty, and neither the capacity nor false positive rate have a significant impact on GPU performance. If workloads emerge that heavily utilize concurrent CPU-GPU threads, the size and performance of this structure will need to be re-evaluated. However if `cacheflush` trapping should become excessive due to an undersized remote directory, page-migration of CPU–GPU shared pages out of GPU memory and into CPU memory can also be employed to reduce pressure on the GPU remote directory.

5.2.2 Improving Selective Caching Performance

Caches have consistently been shown to provide significant performance gains thanks to improved bandwidth and latency. As such, naively bypassing the GPU caches based on the mechanisms described in Section 5.2.1 should be expected to hurt performance. In this subsection we describe three architectural improvements that mitigate the impact of selectively bypassing the GPU caches and provide performance approaching a system with hardware cache coherence.

5.2.2.1 Cacheless Request Coalescing

The first optimization we make to our naive selective caching design is to implement aggressive miss status handling register (MSHR) request coalescing for requests sent to CPU memory, labeled ❶ in Figure 5.2. Despite not having any cache storage for requests from CPU-memory, using MSHR-style request coalescing for read requests can significantly reduce the number of requests made to CPU memory without violating coherency guarantees. Request coalescing works by promoting the granularity of an individual load request (that may be as small as 64 bits) to a larger granularity (typically 128B cache

lines) before issuing the request to the memory system. While this larger request is in-flight, if other requests are made within the same 128B block, then these requests can simply be attached to the pending request list in the corresponding MSHR and no new request is issued to the memory system.

To maintain correctness in a non-caching system, this same coalescing scheme can be utilized, but data that is returned to the coalesced requests for which no pending request is found, must be discarded immediately. Discarding data in this way is similar to self-invalidating coherence protocols, which attempt to minimize invalidation traffic in CC-NUMA systems [70, 71]. Whereas most MSHR implementations allocate their storage in the cache into which the pending request will be inserted, our cache-less request coalescing must have local storage to latch the returned data. This storage overhead is negligible compared to the aggregate size of the on-chip caches that are no longer needed with selective caching.

Table 5.1 shows the fraction of GPU memory requests that can be coalesced by matching them to pre-existing in-flight memory requests. We call request coalescing that happens within a single SM *L1 coalescing* and coalescing across SMs *L1+L2 coalescing*. On average, 35% of memory requests can be serviced via cacheless request coalescing. While a 35% hit rate may seem low when compared to conventional CPU caches, we observe that capturing spatial request locality via request coalescing provides the majority of the benefit of the L1 caches (44.4% hit rate) found in a hardware cache-coherent GPU, shown in Table 2.1.

5.2.2.2 CPU-side Client Cache

Although memory request coalescing provides hit rates approaching that of conventional GPU L1 caches, it still falls short as it cannot capture temporal locality. Selective caching prohibits the GPU from locally caching lines that are potentially shared with the CPU but it does not preclude the GPU from remotely accessing coherent caches located at the CPU. We exploit this opportunity to propose a *CPU-side GPU client cache*, labeled ❷ in Figure 5.2.

To access CPU memory, the GPU must already send a request to the CPU memory controller to access the line. If request coalescing has failed to capture re-use of a cache line, then multiple requests for the same line will be sent to the CPU memory controller causing superfluous transfers across the DRAM pins, wasting precious bandwidth. To reduce this DRAM pressure we introduce a small client cache at the CPU memory controller to service these GPU requests, thereby shielding the DDR memory system from

Workload	L1 Coalescing	L1+L2 Coalescing
backprop	54.2	60.0
bfs	15.8	17.6
btree	69.4	82.4
cns	24.8	28.1
comd	45.7	53.8
kmeans	0.0	0.0
minife	29.0	32.6
mummer	41.9	51.1
needle	0.1	1.8
pathfinder	41.4	45.8
srad_v1	30.8	34.2
xsbench	15.6	18.0
Average	30.7	35.4

Table 5.1: Percentage of memory accesses that can be coalesced into existing in-flight memory requests, when using L1 (intra-SM) coalescing, and L1 + L2 (inter-SM) coalescing.

repeated requests for the same line. Our proposed GPU client cache participates in the CPU coherence protocol much like any other coherent cache on the CPU die, however lines are allocated in this cache only upon request by an off-chip processor, such as the GPU.

This single new cache does not introduce the coherence and interconnect scaling challenges of GPU-side caches, but still provides some latency and bandwidth filtering advantages for GPU accesses. One might consider an alternative where GPU-requested lines are instead injected into the existing last-level cache (LLC) at the CPU. In contrast to an injection approach, our dedicated client cache avoids thrashing the CPU LLC when the GPU streams data from CPU memory (a common access pattern). By placing this client cache on the CPU-side rather than the GPU-side of the CPU–GPU interconnect, we decouple the need to extend the CPU's hardware cache coherence protocol into even one on-die GPU cache. However, because the GPU client cache is located at the CPU-side of the CPU–GPU interconnect, it provides less bandwidth than a GPU-side on-die cache. As described in Chapter II and Figure 2.3 shows, this bandwidth loss may not be performance critical.

5.2.2.3 Variable-size Link Transfers

Conventional memory systems access data at cache line granularity to simplify addressing and request matching logic, improve DRAM energy consumption, and exploit

Workload	Avg. Cacheline Utilization(%)
backprop	85.9
bfs	37.4
btree	78.7
cns	77.6
comd	32.6
kmeans	25.0
minife	91.6
mummer	46.0
needle	39.3
pathfinder	86.6
srad_v1	96.3
xsbench	30.3
Average	60.6

Table 5.2: Utilization of 128B cache line requests where the returned data must be discarded if there is no matching coalesced request.

spatial locality within caches. Indeed, the minimum transfer size supported by DRAM is usually a cache line. Cache line-sized transfers work well when data that was not immediately needed can be inserted into an on-chip cache, but with selective caching, unrequested data transferred from CPU memory must be discarded. Hence, portions of a cache line that were transferred, but not matched to any coalesced access, result in wasted bandwidth and energy.

The effect of this data over-fetch is shown in Table 5.2, where cache line utilization is the fraction of the transferred line that has a pending request when the GPU receives a cache line-sized response from CPU memory. An average cache line utilization of 60% indicates that just 77 out of 128 bytes transferred are actually used by the GPU. 51 additional bytes were transferred across the DRAM interface and CPU–GPU interconnect only to be immediately discarded.

To address this inefficiency, architects might consider reducing the transfer unit for cacheless clients from 128B down to 64 or 32 bytes. While fine-grained transfers improve transfer efficiency by omitting unrequested data, that efficiency is offset by the need for multiple small requests and packetization overhead on the interconnect. For example, in our link implementation, a transfer granularity of 32B achieves at best 66% link utilization (assuming all data is used) due to interconnect protocol overheads, while 128B transfers (again, assuming all data is used) can achieve 88% efficiency.

To maintain the benefit of request coalescing, but reduce interconnect inefficiency, we propose using *variable-size transfer units* on the CPU–GPU interconnect (labeled ❸ in

Figure 5.2). To implement variable-size transfer units at the GPU, we allocate GPU MSHR entries at the full 128B granularity; coalescing requests as described in Section 5.2.2.1. However, when a request is issued across the CPU–GPU interconnect, we embed a bitmask in the request header indicating which 32B sub-blocks of the 128B cache line should be transferred on the return path. While this initial request is pending across the interconnect, if additional requests for the same 128B cache line are made by the GPU, those requests will be issued across the interconnect and their 32B sub-block mask will be merged in the GPU MSHR.

Similar to the GPU-side MSHR, variable sized transfer units require that the CPU-side client cache also maintain pending MSHR masks for requests it receives, if it can not service the requests immediately from the cache. By maintaining this mask, when the DRAM returns the 128B line, only those 32B blocks that have been requested are transferred to the GPU (again with a bitmask indicating which blocks are included). Because there may be both requests and responses in-flight simultaneously for a single 128B line, it is possible that two or more responses are required to fulfill the data requested by a single MSHR; the bitmasks included in each response facilitate this matching. Because GPUs typically perform SIMD lane-level request coalescing within an SM, 32B requests happen to be the minimum and most frequently sized request issued to the GPU memory system. As a result, we do not investigate supporting link transfer sizes smaller than 32 bytes, which would require microarchitectural changes within the GPU SM.

5.2.3 Promiscuous Read-Only Caching

Selective caching supports coherence guarantees by bypassing GPU caches when hardware cache-coherence operations could be needed. Thus far, our selective caching architecture has assumed that the GPU must avoid caching all data homed in CPU memory. We identify that we can loosen this restriction and allow GPU caching of CPU memory, but only if that data can be guaranteed to be read-only by both the CPU and GPU.

Figure 5.3 shows the fraction of data touched by the GPU that is read-only or both read and written, broken down at the OS page (4KB) granularity. In many workloads, we find the majority of the data touched by the GPU is read-only at the OS page level. We examine this data at page granularity because, even without hardware cache coherence, it is possible (though expensive) to guarantee correctness through OS page protection mechanisms entirely in software. Any cache may safely contain data from read-only OS pages. However, if the page is re-mapped as read-write, cached copies of the data at the GPU must be discarded, which will occur as part of the TLB shootdown process triggered

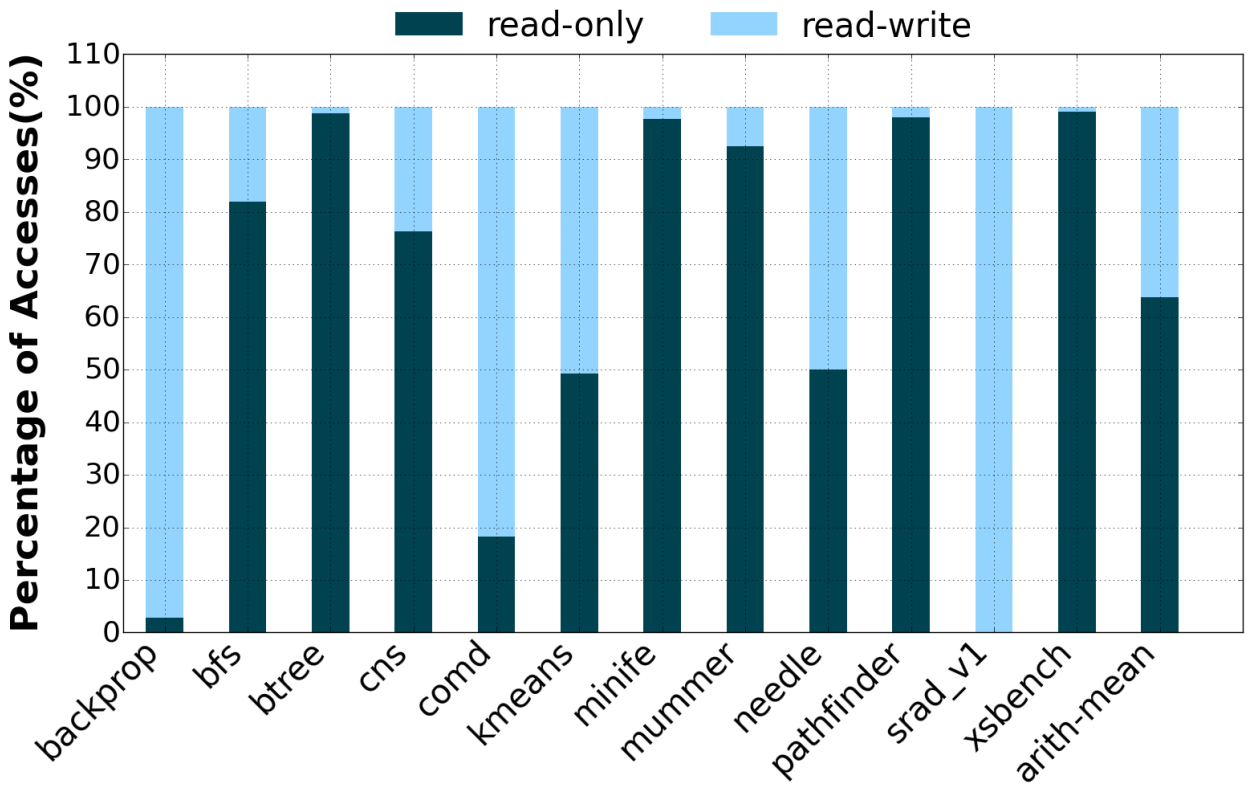


Figure 5.3: Fraction of 4KB OS pages that are read-only and read-write during GPU kernel execution.

by the permission change [72].

We propose that despite lacking hardware cache coherence, selective caching GPUs may choose to implement *promiscuous read-only caching of CPU-memory*, relying on such page level software coherence to provide correctness (labeled ❹ in Figure 5.2). To implement read-only caching, the GPU software run-time system speculatively marks pages within the application as read-only at GPU kernel launch time. It also tracks which pages may have been marked read-only by the application itself to prevent speculation conflicts. With pages speculatively marked as read-only, when the GPU requests pages from the CPU memory, the permissions bit in the TLB entry is checked to determine if lines from this page are cacheable by the GPU despite being homed in CPU memory. Similarly, if the line resides in GPU memory but is marked as cached by the CPU in the remote directory, this line can still be cached locally because it is read-only.

If a write to a read-only page occurs at either the CPU or GPU, a protection fault is triggered. A write by the CPU invokes a fault handler on the faulting core, which marks the line as read/write at the CPU and uncacheable at the GPU. The fault handler then triggers a TLB shootdown, discarding the now stale TLB entry from all CPU and GPU

TLBs. This protection fault typically incurs a 3-5us delay. The next access to this page at a GPU SM will incur a hardware page walk to refetch this PTE, typically adding ¡ 1us to the first access to this updated page.

A faulting write at the GPU is somewhat more complex, as protection fault handlers currently do not run on a GPU SM. Instead, the GPU MMU must dispatch an interrupt to the CPU to invoke the fault handler. That SW handler then adjusts the permissions and shoots down stale TLB entries, including those at the GPU. The CPU interrupt overhead raises the total unloaded latency of the fault to 20us (as measured on NVIDIA's Maxwell generation GPUs). However, only the faulting warp is stalled: the SM can continue executing other non-faulting warps. Once the GPU receives an acknowledgement that the fault handling is complete, it will re-execute the write, incurring a TLB miss and a hardware page walk to fetch the updated PTE entry.

The many-threaded nature of the GPU allows us to largely hide the latency of these permission faults by executing other warps, thereby mitigating the performance impact of the high SW fault latency in nearly all of our workloads. Nevertheless, software page fault handlers are orders of magnitude more expensive than hardware cache-coherence messaging and may erode the benefit of promiscuous read-only caching if permission faults are frequent. We evaluate the performance of promiscuous caching under different software faulting overhead costs in Section 5.4.2.

5.3 Methodology

We evaluate selective caching via simulation on a system containing discrete CPUs and GPUs with DDR4 and GDDR5 memories attached to the CPU and GPU, respectively. We discuss our baseline simulation environment in Chapter 3.2.2.1. To implement various architectural components we modify our framework with simulation parameters shown in Table 5.3. We use bandwidth-aware page placement for all simulations as it has been shown to be the best page placement strategy without requiring any application profiling or program modification [14]. In our simulated system, this page placement results in 20% of the GPU workload data being placed within the CPU-attached memory with 80% residing in the GPU-attached memory.

In our system, the CPU is connected to the GPU via a full duplex CPU–GPU interconnect. The interconnect has peak bandwidth of 90GB/s using 16B flits for both data and control messages with each data payload of up to 128B requiring a single header flit. Thus, for example, a 32B data message will require sending 1 header flit + 2 data flits = 3 flits in total. When simulating request coalescing within the GPU, we use the same num-

Memory System	
CPU Client Cache	512KB, 200 cycle latency
GPU GDDR5	8-channels, 336GB/sec aggregate
CPU DDR4	4-channels, 80GB/sec aggregate
SW Page Faults	16 concurrent per SM
DRAM Timings	RCD=RP=12, RC=40, CL=WR=12
DDR4 Burst Len.	8
CPU–GPU Interconnect	
Link Latency	100 GPU core cycles
Link Bandwidth	90 GB/s Full-Duplex
Req. Efficiency	32B=66%, 64B=80%, 128B=88%

Table 5.3: Parameters for experimental GPGPU based simulation environment.

ber of MSHRs as the baseline configuration but allow the MSHRs to have their own local return value storage in the cacheless request coalescing case. The CPU-side GPU client cache is modeled as an 8-way set associative, write-through, no write-allocate cache with 128B line size of varying capacities shown later in Section 5.4.1.2. The client cache latency is 200 cycles, comprising 100 cycles of interconnect and 100 cycles of cache access latency. To support synchronization operations between CPU and GPU, we augment the GPU MSHRs to support atomic operations to data homed in either physical memory; we assume the CPU similarly can issue atomic accesses to either memory.

To model promiscuous read-only caching, we initially mark all the pages (4kB in our system) in DDR as read-only upon GPU kernel launch. When the first write is issued to each DDR page, the ensuing protection fault invalidates the TLB entry for the page at the GPU. When the faulting memory operation is replayed, the updated PTE is loaded, indicating that the page is uncacheable. Subsequent accesses to the page are issued over the CPU–GPU interconnect. Pages marked read-write are never re-marked read-only during GPU kernel execution. Using the page placement policy described earlier in this section, the GPU is able to cache 80% of the application footprint residing in GPU memory. We vary our assumption for the remote protection fault latency from 20-40us and assume support for up to 16 pending software page protection faults per SM; a seventeenth fault blocks the SM from making forward progress on any warp.

We evaluate results using the Rodinia and United States Department of Energy benchmark suites. We execute the applications under the CUDA 6.0 weak consistency memory model. While we did evaluate workloads from the Parboil [28] suite, we found that these applications have uncharacteristically high cache miss rates, hence even in the hardware cache-coherent case, most memory accesses go to the DRAM. As such, we have cho-

sen to omit these results because they would unfairly indicate that selective caching is performance equivalent to a theoretical hardware cache-coherent GPU. In Section 6.5 we report GPU performance as application throughput, which is inversely proportional to workload execution time.

5.4 Results

We evaluate the performance of selective GPU caching through iterative addition of our three proposed microarchitectural enhancements on top of naive selective caching. We then add promiscuous read-only caching and finally present a sensitivity study for scenarios where the workload footprint is too large for a performance-optimal page placement split across CPU and GPU memory.

5.4.1 Microarchitectural Enhancements

Figure 5.4 shows the baseline performance of naive selective caching compared to a hardware cache-coherent GPU. Whereas performance remains as high as 95% of the baseline for some applications, the majority of applications suffer significant degradation, with applications like `btree` and `comd` seeing nearly an order-of-magnitude slowdown. The applications that are hurt most by naive selective caching tend to be those that have a high L2 cache hit rate in a hardware cache-coherent GPU implementation like `comd` (Table 2.1) or those that are highly sensitive to L2 cache latency like `btree` (Figure 2.3). Prohibiting all caching of CPU memory results in significant over-subscription of the CPU memory system, which quickly becomes the bottleneck for application forward progress, resulting in nearly a 50% performance degradation across our workload suite.

5.4.1.1 Cacheless Request Coalescing

Our first microarchitectural proposal is to implement cacheless request coalescing as described in Section 5.2.2.1. With naive selective caching relying on only the lane-level request coalescer, performance of the system degrades to just 42% of the hardware cache-coherent GPU, despite only 20% of the application data residing in CPU physical memory. Introducing request coalescing improves performance to 74% and 79% of a hardware cache-coherent GPU when using L1 coalescing and L1+L2 coalescing, respectively. This improvement comes from a drastic reduction in the total number of requests issued across the CPU–GPU interconnect and reducing pressure on the CPU memory. Surprisingly `srad_v1` shows a 5% speedup over the hardware cache-coherent GPU when

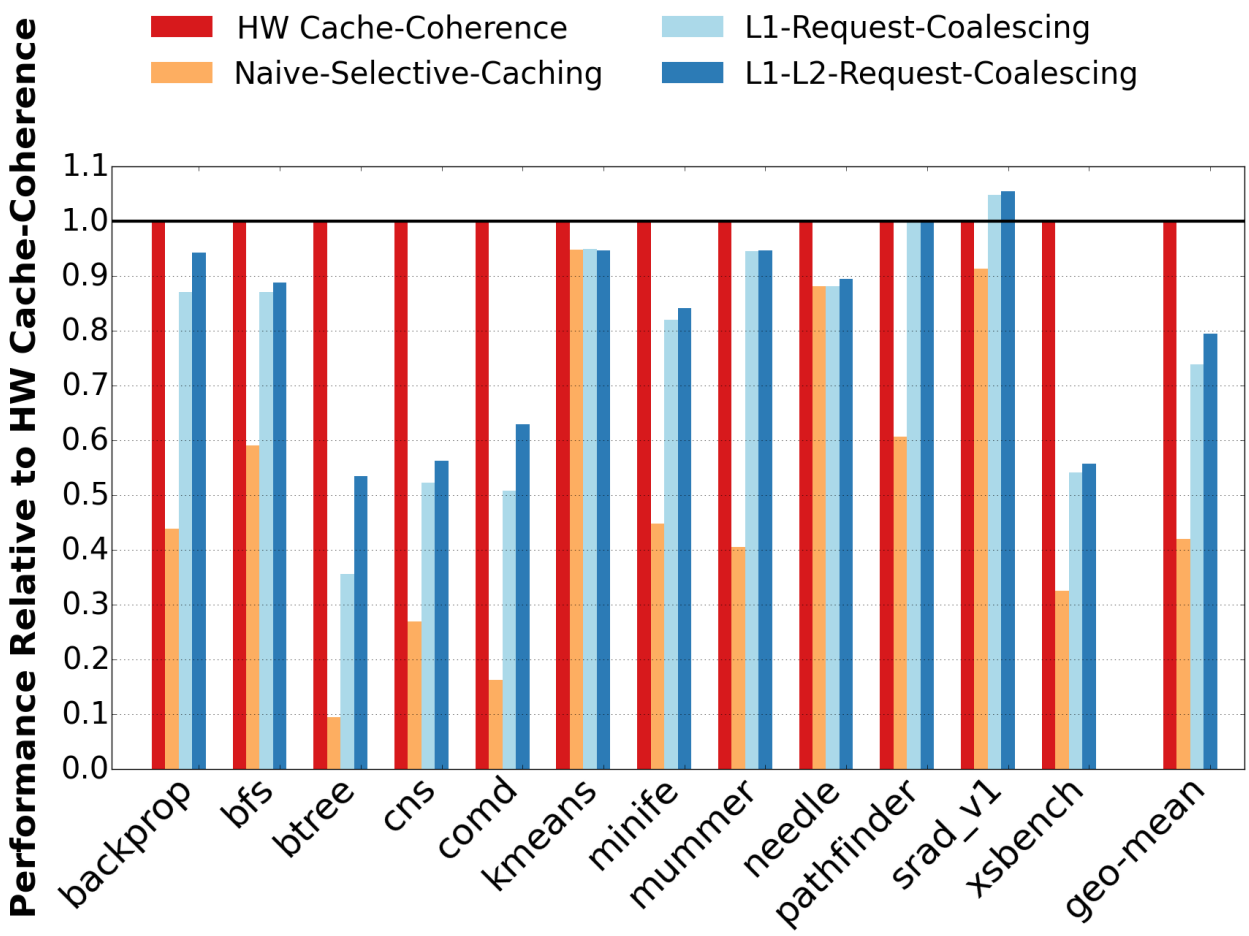


Figure 5.4: GPU performance under selective caching with uncoalesced requests, L1 coalesced requests, L1+L2 coalesced requests.

using L1+L2 request coalescing. `srad_v1` has a large number of pages that are written without first being read, thus the CPU DRAM system benefits from the elimination of reads that are caused by the write-allocate policy in the baseline GPU's L2 cache. Because the request coalescing hit rates, shown in Table 5.1, lag behind the hardware cached hit rates, selective caching still places a higher load on the interconnect and CPU memory than a hardware cache-coherent GPU, which translates into the 21% performance reduction we observe when using selective caching with aggressive request coalescing.

5.4.1.2 CPU-side Client Cache

Whereas request coalescing captures much of the spatial locality provided by GPU L1 caches, it cannot capture any long distance temporal locality. Figure 5.5 shows the performance differential of adding our proposed CPU-side client cache to L1+L2 request

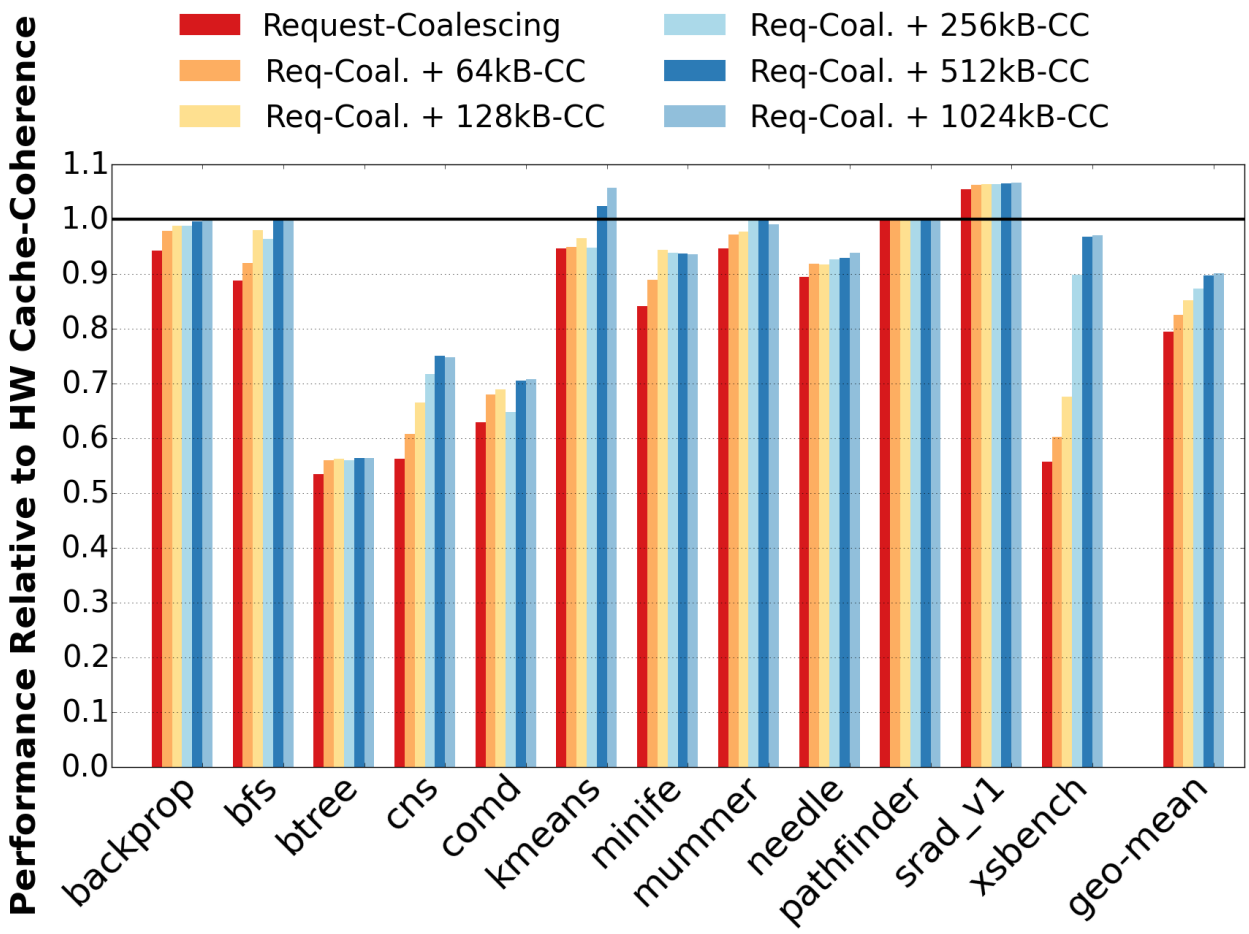


Figure 5.5: GPU performance with selective caching when combining request coalescing with on CPU-side caching for GPU clients at 64KB–1MB cache capacities. (CC: Client-Cache)

coalescing within the selective caching GPU. This GPU client cache not only reduces traffic to CPU DRAM from the GPU, but also improves latency for requests that hit in the cache and provides additional bandwidth that the CPU–GPU interconnect may exploit. We observe that performance improvements scale with client cache size up to 512KB before returns diminish. Combining a 512KB, 8-way associative client cache with request coalescing improves performance of our selective caching GPU to within 90% of the performance of a hardware cache-coherent GPU. Note that `btree` only benefits marginally from this client cache because accessing the client cache still requires a round-trip interconnect latency of 200ns (Section 5.3). `btree` is highly sensitive to average memory access latency (Figure 2.3), which is not substantially improved by placing the client cache on the CPU-die rather than the GPU-die.

The size of an on-die CPU client cache is likely out of the hands of GPU architects, and

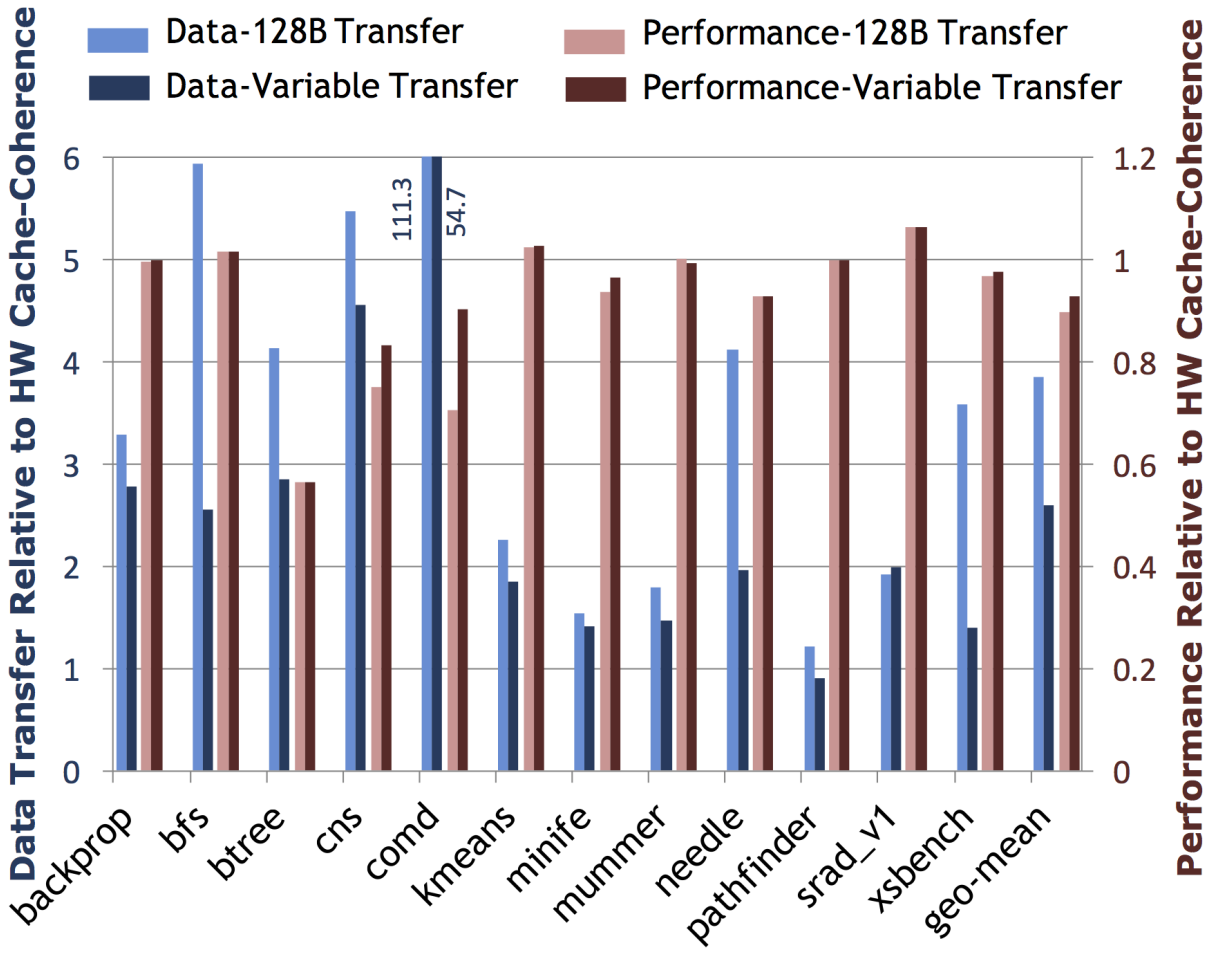


Figure 5.6: GPU data transferred across CPU-GPU interconnect (shown left y-axis) and performance (shown right y-axis) for 128B cache line-size link transfers and variable-size link transfers respectively.

for CPU architects allocating on-die resources for an external GPU client may seem an unlikely design choice. However, this client cache constitutes only a small fraction of the total chip area of modern CPUs (0.7% in 8-core Xeon E5 [73]) and is the size of just one additional private L2 cache within the IBM Power 8 processor. Much like processors have moved towards on-die integration of PCIe to provide improved performance with external peripherals, we believe the performance improvements due to this cache are significant enough to warrant integration. For CPU design teams, integrating such a cache into an existing design is likely easier than achieving performance by extending coherence protocols into externally developed GPUs. The GPU client cache also need not be specific to just GPU clients, other accelerators such as FPGAs or spatial architectures [74, 75] that will be integrated along-side a traditional CPU architecture will also likely benefit from such a client cache.

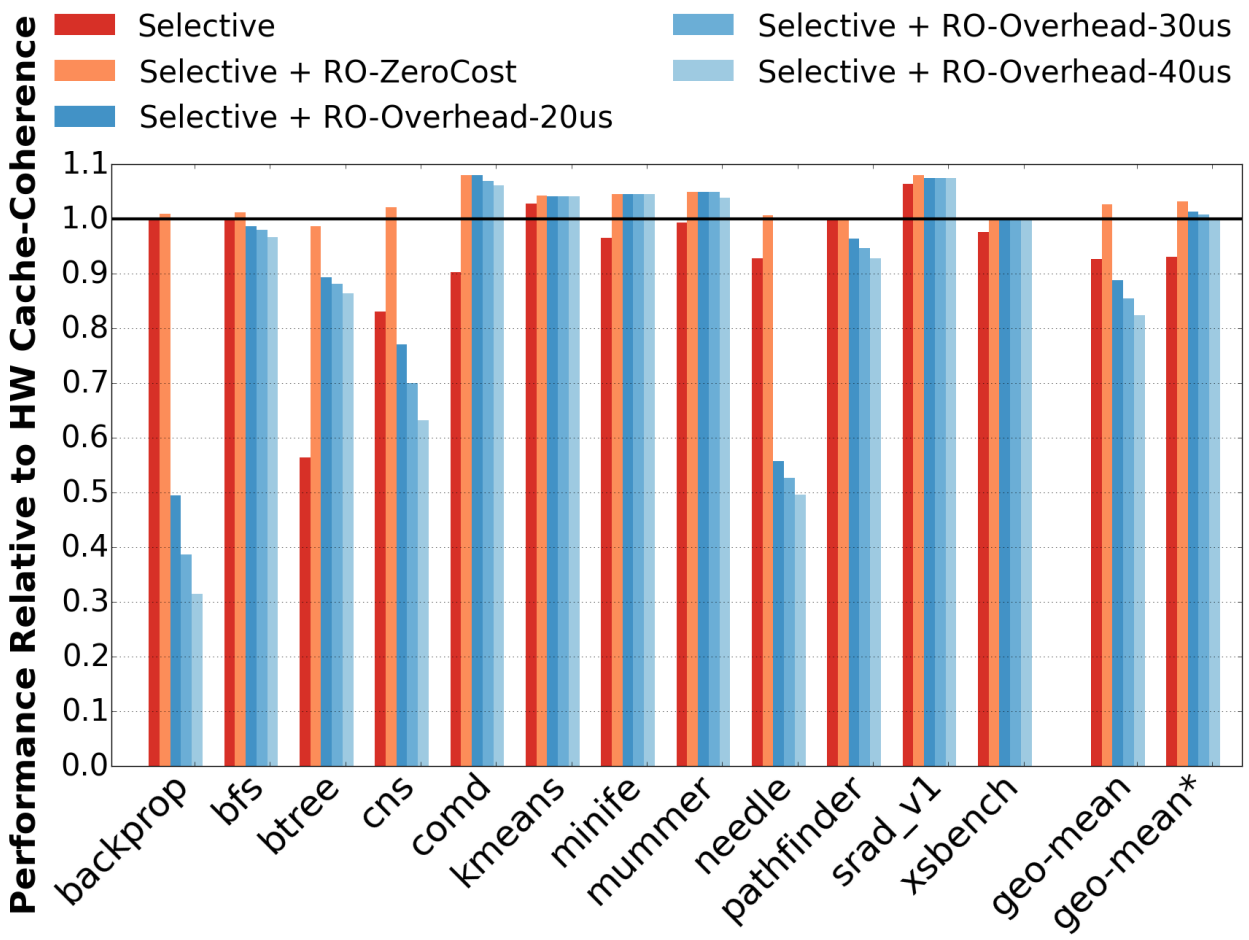


Figure 5.7: GPU performance when using Selective caching (Request-Coalescing + 512kB-CC + Variable-Transfers) combined with read-only based caching. geo-mean*: Geometric mean excluding `backprop`, `cns`, `needle`, where read-only caching would be switched-off. (RO: Read-Only)

5.4.1.3 Variable-size Link Transfers

Request coalescing combined with the CPU client cache effectively reduce the pressure on the CPU DRAM by limiting the number of redundant requests that are made to CPU memory. The CPU client cache exploits temporal locality to offset data overfetch that occurs on the DRAM pins when transferring data at cache line granularity, but does not address CPU–GPU interconnect transfer inefficiency. To reduce this interconnect over-fetch, we propose variable-sized transfer units (see Section 5.2.2.3). The leftmost two bars for each benchmark in Figure 5.6 show the total traffic across the CPU–GPU interconnect when using traditional fixed 128B cache line requests and variable-sized transfers, compared to a hardware cache-coherent GPU. We see that despite request coalescing, our selective caching GPU transfers nearly 4 times the data across the CPU–GPU intercon-

nect than the hardware cache-coherent GPU. Our variable-sized transfer implementation reduces this overhead by nearly one third to just 2.6x more interconnect traffic than the hardware cache-coherent GPU.

This reduction in interconnect bandwidth results in performance gains of just 3% on average, despite some applications like `comd` showing significant improvements. We observe that variable-sized transfers can significantly improve bandwidth utilization on the CPU–GPU interconnect but most applications remain performance-limited by the CPU memory bandwidth, not the interconnect itself. When we increase interconnect bandwidth by 1.5x without enabling variable-sized requests, we see an average performance improvement of only 1% across our benchmark suite. Variable-sized requests are not without value, however; transferring less data will save power or allow this expensive off-chip interface to be clocked at a lower frequency, but evaluating the effect of those improvements is beyond the scope of this work.

5.4.2 Promiscuous GPU Caching

By augmenting selective caching with request coalescing, a GPU client cache, and variable-sized transfers, we achieve performance within 93% of a hardware cache-coherent GPU. As described in Section 5.2.3, the GPU can be allowed to cache CPU memory that is contained within pages that are marked as read-only by the operating system. The benefit of caching data from such pages is offset by protection faults and software recovery if pages promiscuously marked as read-only and cached by the GPU are later written. Figure 5.7 (RO-ZeroCost) shows the upper bound on possible improvements from read-only caching for an idealized implementation that marks all pages as read-only and transitions them to read-write (and thus uncacheable) without incurring any cost when executing the required protection fault handling routine. In a few cases, this idealized implementation can outperform the hardware cache-coherent GPU because of the elimination of write allocations in the GPU caches, which tend to have little to no reuse.

We next measure the impact of protection fault cost, varying the unloaded fault latency from 20us to 40us (see Figure 5.7) which is available on today's GPU implementations. While a fault is outstanding, the faulting warp and any other warp that accesses the same address are stalled; but, other warps may proceed, mitigating the impact of these faults on SM forward progress. The latency of faults can can be hidden if some warps executing on an SM are reading this or other pages. However, if all warps issue writes at roughly the same time, the SM may stall due to a lack of schedulable warps or MSHR capacity to track pending faults. When accounting for fault overheads, our selective caching GPU with

promiscuous read-only caching achieves only 89% of the performance of the hardware cache-coherent GPU.

When using a 20us fault latency, we see that 7 of 12 workloads exhibit improvement from read-only caching and that `btree` sees a large 35% performance gain from promiscuous read-only caching as it benefits from improvements to average memory access latency. In contrast, three workloads, `backprop`, `cns`, and `needle`, suffer considerable slowdowns due to exposed protection fault latency. These workloads tend to issue many concurrent writes, exhausting the GPUs ability to overlap execution with the faults. For such workloads, we advocate disabling promiscuous read-only caching in software (e.g., via a mechanism that tracks the rate of protection faults, disabling promiscuous read-only caching when the rate exceeds a threshold).

In summary, the effectiveness of promiscuous read-only caching depends heavily on the latency of protection faults and the GPU microarchitecture's ability to overlap the execution of non-faulting warps with those faults, which can vary substantially across both operating systems and architectures. In systems where the fault latency is higher than the 20us (as measured on current NVIDIA systems), more judicious mechanisms must be used to identify read-only pages (e.g., explicit hints from the programmer via the `mprotect` system call.)

5.4.3 Discussion

One use case in the future may be that GPU programmers will size their application's data to extend well beyond the performance-optimal footprint in CPU and GPU memory. With excess data spilling over into the additional capacity provided by the CPU memory, performance bottlenecks will shift away from the GPU towards the CPU memory system. In such cases, the GPU caching policy for CPU memory will come under additional pressure due to the increased traffic skewed towards CPU memory.

To understand how selective caching affects performance under such a scenario, we evaluate a situation wherein the application data has been sized so that 90% of the footprint resides in CPU memory and just 10% can fit within GPU memory, as compared to the nearly inverse performance-optimal 20%-80% ratio. Figure 5.8 shows the performance of this memory-capacity-constrained case relative to the baseline optimal ratio. We see that naive selective caching and our proposed enhancements follow the same trend of performance improvements shown previously in Section 6.5. Because this scenario is primarily limited by the CPU memory system, we see that in some cases the client cache and variable sized transfer interconnect optimizations can actually outper-

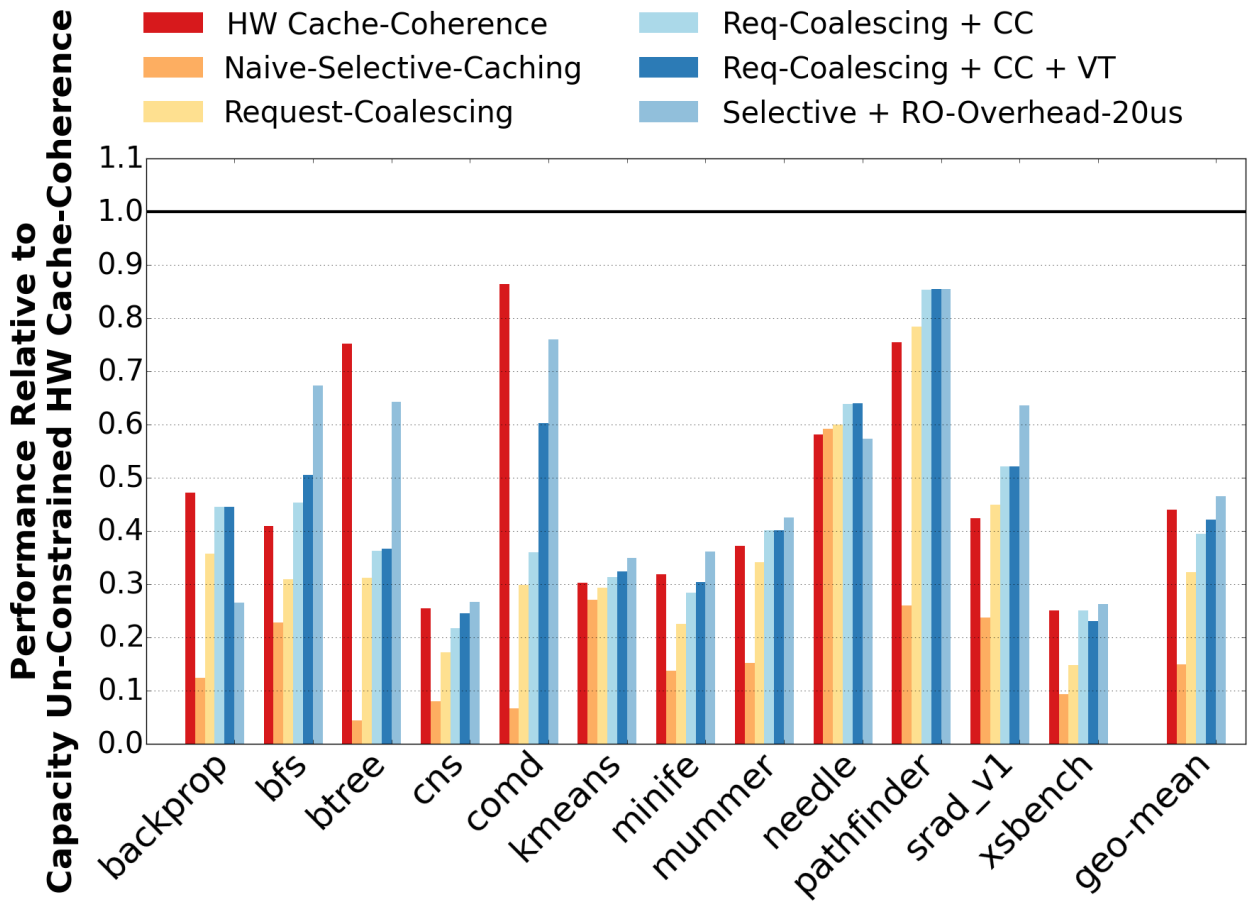


Figure 5.8: GPU performance under memory capacity constraints. (CC: Client-Cache, VT: Variable-sized Transfer Units)

form the hardware cache-coherent GPU due to a reduction in data overfetch between the CPU memory and the GPU client. To validate our observation, we added the same client cache and variable transfers to the hardware cache-coherent baseline configuration and saw an average speedup of 4.5%. Whereas the absolute performance achieved, compared to a performance-optimal memory footprint and allocation, may not always be compelling, should software designers chose to partition their problems in this way, we believe selective caching will continue to perform as well as a hardware cache-coherent GPU.

In this work, we have primarily investigated a system where bandwidth-aware page placement provides an initial page placement that has been shown to have optimal performance [14]. Bandwidth-aware page placement is based on the premise that the GPU will place pressure on the CPU and GPU memory system in proportion to the number of pages placed in each memory. Proposals, like selective caching, that change the on-chip caching policy of the GPU can cause dramatic shifts in the relative pressure placed

on each memory system, effectively changing the bandwidth-optimal placement ratio. Although we do not evaluate this phenomenon in this work, balancing initial page placement with dynamic page migration to help compensate for the lack of on-chip caching is an area that needs further investigation.

5.5 Conclusion

In this chapter, we demonstrate that CPUs and GPUs do not need to be hardware cache-coherent to achieve the simultaneous goals of unified shared memory and high GPU performance. Our results show that *selective caching* with request coalescing, a CPU-side GPU client cache, variable-sized transfer units can perform within 93% of a cache-coherent GPU for applications that do not perform fine grained CPU–GPU data sharing and synchronization. We also show that promiscuous read-only caching benefits memory latency sensitive applications using OS page-protection mechanisms rather than relying on hardware cache coherence. Selective caching does not needlessly force hardware cache coherence into the GPU memory system, allowing decoupled designs that can maximize CPU and GPU performance, while still maintaining the CPU's traditional view of the memory system.

CHAPTER VI

Thermostat: Application-transparent Page Management for Two-tiered Main Memory

6.1 Introduction

Upcoming memory technologies, such as Intel/Micron's recently-announced 3D XPoint memory [24], are projected to be denser and cheaper per bit than DRAM while providing the byte-addressable load-store interface of conventional main memory. Improved capacity and cost per bit come at the price of higher access latency, projected to fall somewhere in the range of 400ns to several microseconds [24] as opposed to 50–100ns for DRAM. The impending commercial availability of such devices has renewed interest in two-tiered physical memory, wherein part of a system's physical address space is implemented with the slower, cheaper memory technology [76, 77].

Slow memory can result in a net cost win if the cost savings of replaced DRAM outweigh cost increase due to reduced program performance or by enabling a higher peak memory capacity per server than is economically viable with DRAM alone. To realize cost savings, in this study, we set an objective of at most 3% performance degradation relative to a DRAM-only system.

However, making effective, transparent use of slow memory to reduce cost without substantial performance loss is challenging. Any memory placement policy must estimate the performance degradation associated with placing a given memory page in slow memory, which in turn requires some method to gauge the *page access rate*. Lack of accurate page access rate tracking in contemporary x86 hardware makes this task challenging. Moreover, as we will show, naive policies to place pages into slow memory based on existing hardware-maintained *Accessed* bits are insufficient and can lead to severe performance degradations.

Making slow memory usage application transparent is particularly critical for cloud

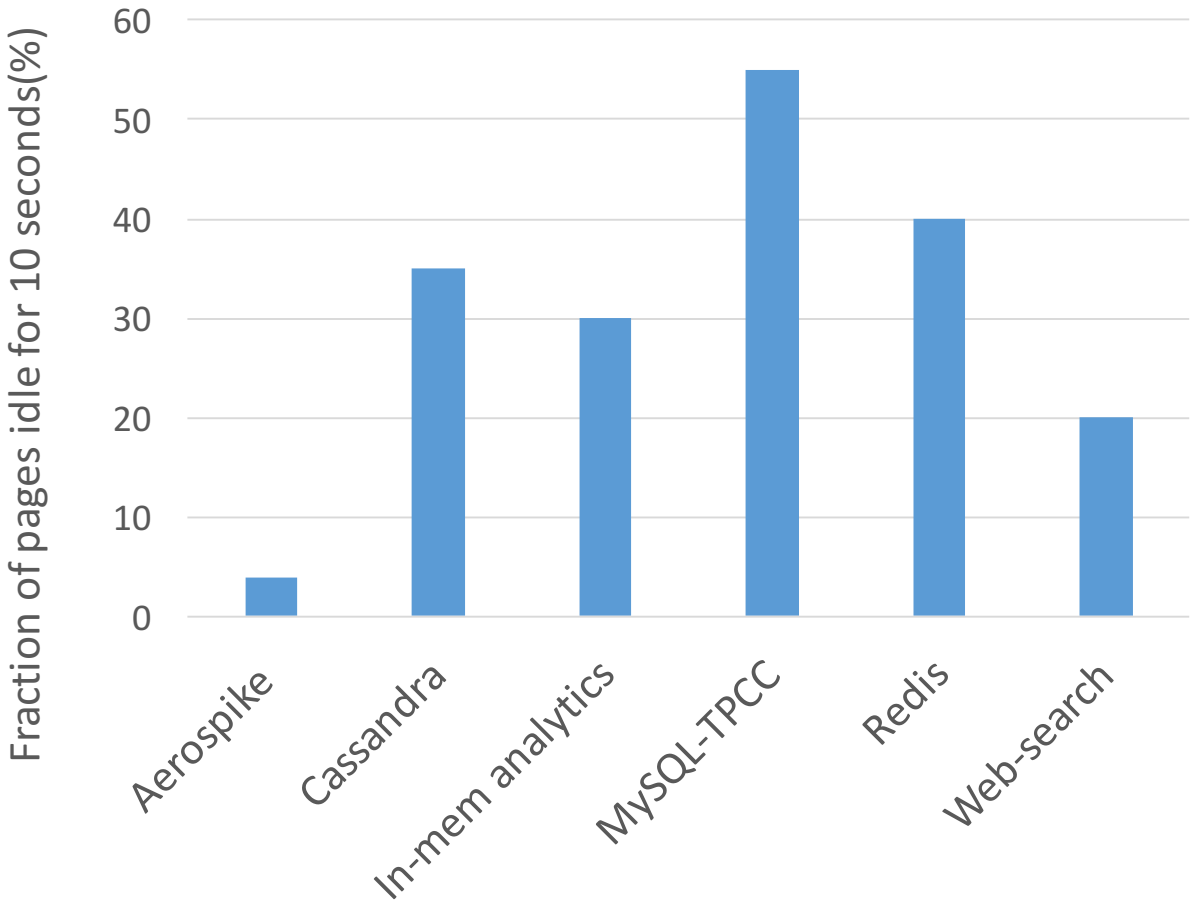


Figure 6.1: Fraction of 2MB pages idle/cold for 10s detected via per-page *Accessed* bits in hardware. Note that this technique cannot estimate the page access rate, and thus cannot estimate the performance degradation caused by placing these pages in slow memory (which exceeds 10% for Redis).

computing environments, where the cloud provider may wish to transparently substitute cheap memory for DRAM to reduce provisioning cost, but has limited visibility and control of customer applications. Relatively few cloud customers are likely to take advantage of cheaper-but-slower memory technology if they must redesign their applications to explicitly allocate and manage hot and cold memory footprints. A host-OS-based cold memory detection and placement mechanism is a natural candidate for such a system. Figure 6.1 shows the amount of data idle for 10s as detected at runtime by an existing Linux mechanism to monitor hardware-managed *Accessed* bits in the page tables for various cloud applications. We observe that substantial cold data (more than 50% for MySQL) can be detected by application-transparent mechanisms.

However, there has been little work on providing performance degradation guaran-

tees in the presence of page migration to slow memory [78]. Furthermore, prior work on two-tiered memory has assumed migration/paging at 4KB page granularity [77, 76]. However, huge pages (2MB pages) are now ubiquitous and critical, especially for cloud platforms where virtualization magnifies the costs of managing 4KB pages. We observe performance improvements as high as 30% from huge pages under virtualization (Table 6.1). Our proposal, *Thermostat*, manages two-tiered main memory transparently to applications while preserving the benefits of huge pages and dynamically enforcing limits on performance degradation (e.g., limiting slowdown to 3%). We will show that Thermostat is huge-page-aware and can place/migrate 4KB and huge pages while limiting performance degradation within a target bound.

Prior work has considered two approaches to two-tiered memory: (i) a paging mechanism [38, 36], wherein accesses to slow memory invoke a page fault that must transfer data to fast memory before an access may proceed, and (ii) via a migration mechanism (as in cache coherent NUMA multiprocessors) [79], wherein no software fault is required. In the latter scenario, a migration mechanism seeks to shuffle pages between tiers to maximize fast-memory accesses. Dulloor et al. [76] have described a programmer-guided data placement scheme in NVRAM. Such techniques are inapplicable when cloud customers run third-party software and do not have access to source code. Li et al. [78] describe a hardware-based technique to accurately gauge the impact of moving a page from NVRAM to DRAM. However, such hardware requires significant changes to contemporary x86 architecture. In contrast, Thermostat does not require *any* additional hardware support apart from the availability of slow memory.

To provide a low overhead cold-page detection mechanism, Thermostat continuously samples a small fraction of pages and estimates page access rate by spatial extrapolation (described in Section 6.3.2). This strategy makes Thermostat both low overhead and fast-reacting. The single *Accessed* bit per page provided by hardware is insufficient to distinguish hot and cold pages with sufficiently low overhead. Instead, Thermostat uses TLB misses as a proxy for LLC misses as they can be tracked in the OS through reserved bits in the PTE, allowing page access rates to be estimated at low overhead. Finally, Thermostat employs a correction mechanism that rapidly detects and corrects mis-classified cold pages (e.g., due to time-changing access patterns).

We implement Thermostat in Linux kernel version 4.5 and evaluate its effectiveness on representative cloud computing workloads running under KVM virtualization. As the cheaper memory technologies we target – such as Intel/Micron's 3D XPoint – are not yet commercially available, our evaluation emulates slow memory using a software technique that triggers translation faults for slow memory pages, yielding a 1us average access

	Performance gain
Aerospike	6%
Cassandra	13%
In-memory Analytics	8%
MySQL-TPCC	8%
Redis	30%
Web-search	No difference

Table 6.1: Throughput gain from 2MB huge pages under virtualization relative to 4KB pages on both host and guest.

latency.

In summary, we make the following contributions:

- We propose an online low-overhead mechanism for estimating the performance degradation due to placing a particular page in slow memory.
- We use this mechanism in an online, huge-page-aware hot/cold page classification system that *only* requires a target maximum slowdown as input.
- We propose an online method to detect mis-classifications and rectify them, thereby minimizing the impact of such mis-classifications on application throughput.
- By emulating slow memory in software, we demonstrate that Thermostat can migrate up to 50% of cloud application footprints to slow memory with a 3% slowdown, reducing memory provisioning cost up to 30%.

6.2 Motivation

We briefly motivate the potential for dual-technology main memory and the importance of huge pages under virtualized execution.

6.2.1 Shifting cold data to cheap memory

For performance-sensitive and high-footprint cloud applications, it is unlikely that cheaper-but-slower memory technologies, such as Intel/Micron's 3D XPoint memory [24], will entirely supplant DRAM main memory. An increase in memory access latency by even a small multiple (e.g., to 500ns) will result in drastic throughput losses, as the working set of these applications typically greatly exceeds cache capacities. Because DRAM accounts

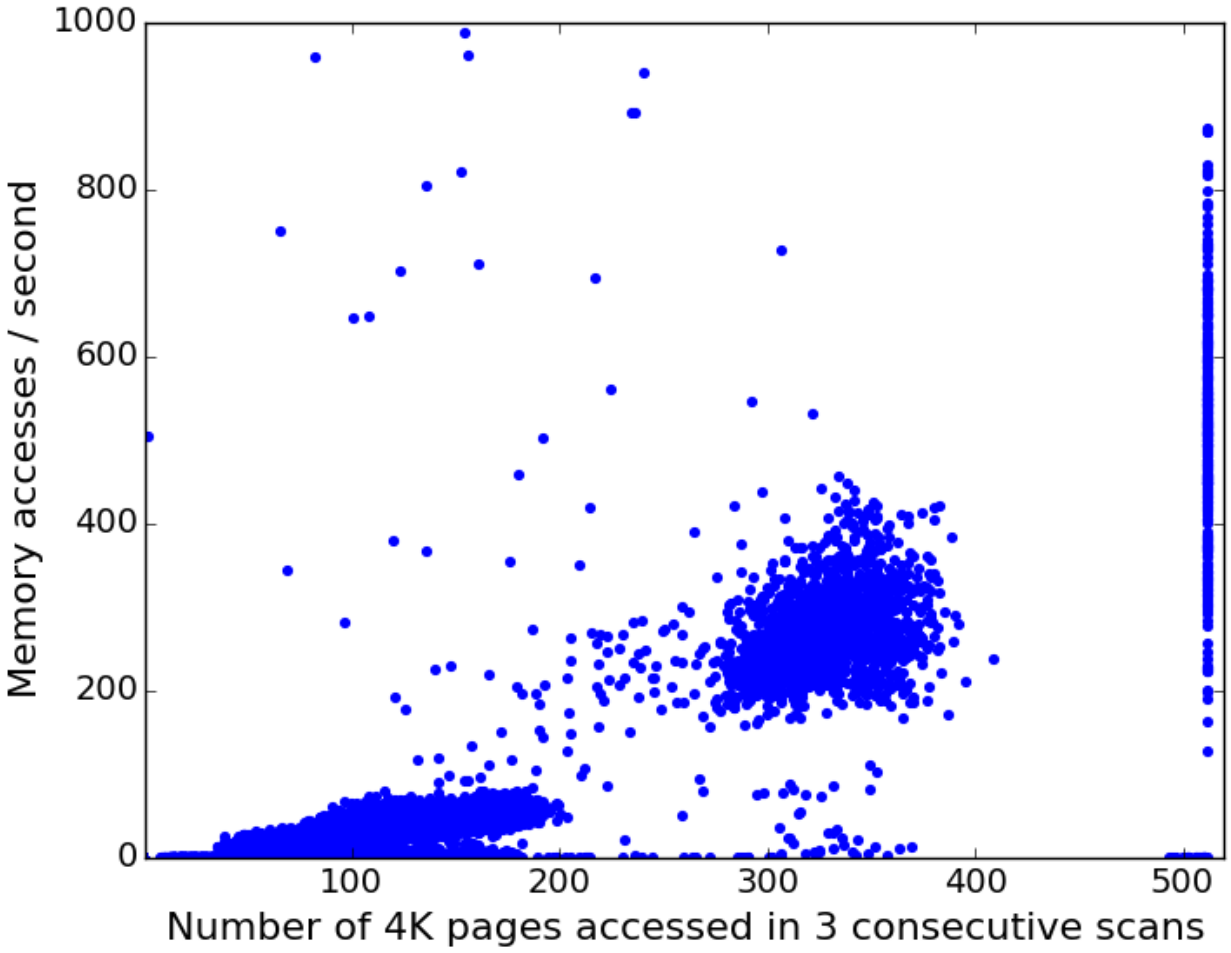


Figure 6.2: Memory access rate vs. hardware *Accessed* bit distribution of 4KB regions within 2MB pages for Redis. The single hardware *Accessed* bit per page does not correlate with memory access rate per page, and thus cannot estimate page access rates with low overhead.

for only a fraction of total system cost, the net cost of the throughput loss will greatly outweigh the savings in main memory cost.

However, cloud applications typically have highly skewed access distributions, where a significant fraction of the application's footprint is infrequently accessed [80]. These rarely accessed pages can be mapped to slow memory without significantly impacting performance. We refer to this infrequently accessed data as "cold" data.

To distinguish cold pages from frequently accessed hot pages, existing mechanisms exploit the *Accessed* bit in the PTE (set by the hardware each time the PTE is accessed by a page walk) [81, 82, 83]. We investigated one such existing cold-page detection framework, `kstaled` [81]. However, we find that the single accessed bit per page is insufficient to distinguish hot and cold 2MB huge pages with sufficiently low overhead. To

detect a page access, `kstaled` must clear the *Accessed* bit and flush the corresponding TLB entry. However, distinguishing hot and cold pages requires monitoring (and hence, clearing) accessed bits at high frequency, resulting in unacceptable slowdowns.

Figure 6.2 illustrates why hot and cold 2MB huge pages cannot be distinguished by temporally sampling the hardware-maintained access bits at the highest possible rate that can meet our tight performance degradation target (3% slowdown), using Redis as an example workload. The hardware can facilitate monitoring at 4KB granularity by temporarily splitting a huge page and monitoring the accessed bits of the 512 constituent 4KB pages (monitoring at 2MB granularity without splitting provides even worse hot/cold differentiation [83]). The horizontal axis represents the number of detected "hot" 4KB regions in a given 2MB page when monitoring at the maximum frequency that meets our slowdown target. Here "hot" refers to pages that were accessed in three consecutive scan intervals. The vertical axis represents the ground-truth memory access rate for each 2MB page. (We describe our methodology for measuring memory access rate in Section 6.3.3). The key take-away is that the scatter plot is highly dispersed—the spatial frequency of accesses within a 2MB page is poorly correlated with its true access rate. Conversely, performance constraints preclude monitoring at higher temporal frequency. Hence, mechanisms that rely solely on *Accessed* bit scanning cannot identify cold pages with low overhead.

6.2.2 Benefits of transparent huge pages

Intel's IA-64 x86 architecture mandates a 4-level page table structure for 4KB memory pages. So, a TLB miss may incur up to four memory accesses to walk the page table. Under virtualization, with two-dimensional page table walks (implemented in Intel's Extended Page Tables and AMD's Nested Page Tables), the cost of a page walk can be as high as 24 memory accesses [84, 85]. When memory is mapped to a 2MB huge page in both the guest and host, the worst-case page walk is reduced to 15 accesses, which significantly lowers the performance overhead of virtualization. Moreover, 2MB huge pages increase the TLB reach and improve the cacheability of intermediate levels of the page tables, as fewer total entries compete for cache capacity.

Table 6.1 shows the performance benefit of using huge pages via Linux's Transparent Huge Page (THP) mechanism. We compare throughput of various cloud applications where both the guest and host employ 2MB huge pages against configurations with transparent huge pages disabled (i.e., all pages are 4KB). We observe significant throughput benefits as high as 30% for Redis. Previous literature has also reported performance

benefits of huge pages [83, 86, 87]. From these results, it is clear that huge pages are essential to performance, and any attempt to employ a dual-technology main memory must preserve the performance advantages of huge pages. For this reason, we only evaluate Thermostat with THP enabled at both host and guest.

6.3 Thermostat

We present Thermostat, an application-transparent huge-page-aware mechanism to detect cold pages during execution. The input to Thermostat is a user-specified tolerable slowdown (3% in our evaluation) incurred as a result of Thermostat's monitoring and due to accesses to data shifted to slow memory. Thermostat periodically samples a fraction of the application footprint and uses a page poisoning technique to estimate the access rate to each page with tightly controlled overhead. The estimated page access rate is then used to select a set of pages to place in cold memory, such that their aggregate access rate will not result in slowdown exceeding the target degradation. These cold pages are then continually monitored to detect and rapidly correct any mis-classifications or behavior changes. In the following sections, we describe the Thermostat in more detail.

6.3.1 Overview

We implement Thermostat in the Linux 4.5 kernel. Thermostat can be controlled at runtime via the Linux memory control group (cgroup) mechanism [88]. All processes in the same cgroup share Thermostat parameters, such as the sampling period and maximum tolerable slowdown.

The Thermostat mechanism comprises four components: (i) a sampling mechanism that randomly selects a subset of pages for access-rate monitoring, (ii) a monitoring mechanism that counts accesses to sampled pages while limiting maximum overhead, (iii) a classification policy to select pages to place in slow memory, and (iv) a mechanism to monitor and detect mis-classified pages or behavior changes and migrate pages back to conventional (fast) memory.

The key challenge that Thermostat must address is the difficulty of discerning the access rates of 2MB pages at a sufficiently low overhead while still responding rapidly to changing workload behaviors. Tracking the access rate of a page is a potentially expensive operation if the page is accessed frequently. Hence, to bound the performance impact of access rate monitoring, only a small fraction of the application footprint may be monitored at any time. However, sampling only a small fraction of the application footprint

leads to a policy that adapts only slowly to changes in memory behavior.

6.3.2 Page sampling

Sampling a large number of huge pages is desirable as it leads to quick response to time-varying workload access patterns. But, it can lead to a high performance overhead, since, as explained in Section 6.3.3, each TLB miss to a sampled page incurs additional latency for OS fault handling. To tightly control application performance slowdown, *we split a random sample of huge pages (5% in our case) into 4KB pages, and poison only a fraction of these 4KB pages in each sampling interval*. Below, we detail the strategy used to select which 4KB pages to poison, and how we estimate the total access rate from the access rates of the sample.

A simple strategy to select 4KB pages from a set of huge pages is to select K random 4KB pages, for some fixed K ($K = 50$ in our evaluation). However, when only a few 4KB pages in a huge page are hot, this naive strategy may fail to sample them, and thus deem the overall huge page to have a low access rate. To address this shortcoming, our mechanism monitors page access rates in two steps. We first rely on the hardware-maintained *Accessed* bits to monitor all 512 4KB pages and identify those with a non-zero access rate. We then monitor a sample of these pages using our more costly software mechanism to accurately estimate the aggregate access rate of the 2MB page. With our strategy, *only 0.5% of memory is sampled at any time*, which makes the performance overhead due to sampling $<$ 1%.

To compute the aggregate access rate at 2MB granularity from the access rates of the sampled 4KB pages, we scale the observed access rate in the sample by the total number of 4KB pages that were marked as accessed. The monitored 4KB pages comprise a random sample of accessed pages, while the remaining pages have a negligible access rate.

6.3.3 Page access counting

Current x86 hardware does not support access counting at a per-page granularity. Thus, we design a software-only solution to track page access rates with very low overhead ($<$1%) by utilizing PTE reserved bits. In Section 6.6.1, we discuss two extensions to existing x86 mechanisms that might enable lower overhead page-access counting.

To approximate the number of accesses to a page, we use BadgerTrap, a kernel extension for intercepting TLB misses [89]. When a page is sampled for access counting, Thermostat poisons its PTE by setting a reserved bit (bit 51), and then flushes the PTE

from the TLB. The next access to the page will incur a hardware page walk (due to the TLB miss) and then trigger a protection fault (due to the poisoned PTE), which is intercepted by BadgerTrap. BadgerTrap's fault handler unpoisons the page, installs a valid translation in the TLB, and then repoisons the PTE. By counting the number of BadgerTrap faults, we can estimate the number of TLB misses to the page, which we use as a proxy for the number of memory accesses.

Note that our approach assumes that the number of TLB misses and cache misses to a 4KB page are similar. For hot pages, this assertion does not hold. However, Thermostat has no need to accurately estimate the access rate to hot pages; it is sufficient to know that they are hot. Conversely, for cold pages, nearly all accesses incur both TLB and cache misses as there is no temporal locality for such accesses, and, therefore, tracking TLB misses is sufficient to estimate the page access rate. We validated our approach by measuring the TLB miss rates (resulting in page-walks) and last-level cache miss rates for our benchmark suite using hardware performance counters via the Linux `perf` utility. For pages we identify as cold, the TLB miss rate is typically higher (but always within a factor of two) of the last-level cache miss rate measured without BadgerTrap, indicating that our approach is reasonable.

6.3.4 Page classification

Classifying pages as hot or cold is governed by the user-specified maximum tolerable slowdown (without such a threshold, one can simply declare all pages cold and call it a day). To select cold pages, we use the estimated access rates of each (huge) page.

We translate a tolerable slowdown of $x\%$ to an access rate threshold in the following way. Given A accesses to slow memory in one second, the total time consumed by slow memory accesses is At_s, where t_s is the access latency of the slow memory. Thus, a slowdown threshold of $x\%$ can be translated to an access rate threshold of $\frac{x}{100t_s}$ per second. If a fraction f of the total huge pages were sampled, we assign pages to the slow memory such that their aggregate estimated access rate does not exceed $f\frac{x}{100t_s}$. We sort the sampled huge pages in increasing order of their estimated access rates, and then place the coldest pages in slow memory until the total access rate reaches the target threshold.

This simple approach governs the access rate to slow memory to avoid the user-specified degradation target. In Figure 6.3, we show slow memory access rate averaged over 30 seconds for our benchmark suite, assuming 1us slow memory access latency and 3% tolerable slowdown (we discuss detailed methodology in Section 6.4). We observe

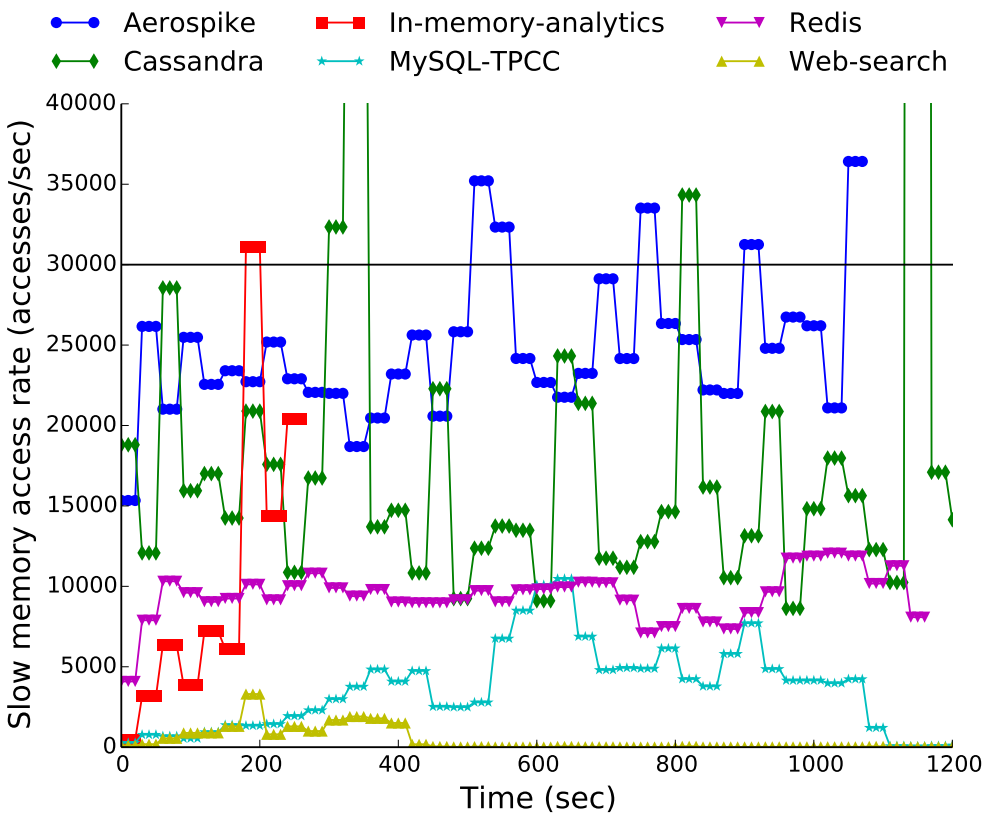


Figure 6.3: Slow memory access rate over time. For a 3% tolerable slowdown and 1us slow memory access latency, the target slow memory access rate is 30K accesses/sec. Thermostat tracks this 30K accesses/sec target. Note that different benchmarks have different run times; we plot the x-axis for 1200 seconds.

that Thermostat keeps the slow memory access rate close to the target 30K accesses/sec. For Aerospike and Cassandra slow memory access rate temporarily exceeds 30K accesses/sec but is brought back below 30K accesses/sec by mis-classification detection, discussed next in Section 6.3.5.

6.3.5 Correction of mis-classified pages

Since we estimate the access rate of a huge page based on the access rates of only a few (not more than 50, as described in Section 6.3.2) 4KB pages, there is always some probability that some hot huge pages will be mis-classified as cold due to sampling error. Such mis-classifications are detrimental to application performance, since the interval between successive samplings of any given huge page can be fairly long. To address this issue, we track the number of accesses being made to each cold huge page, using the software mechanism mentioned in Section 6.3.3. Since the access rate to these pages is

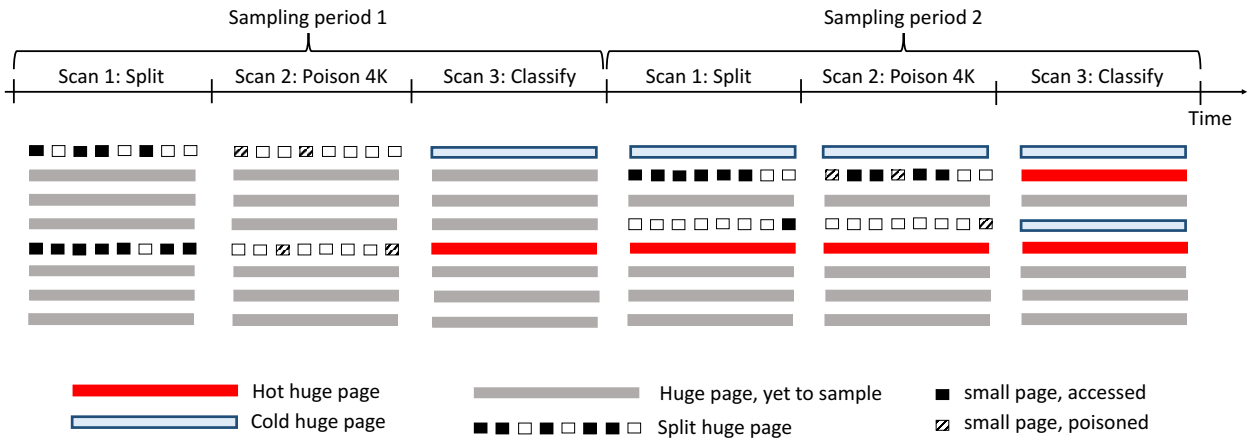


Figure 6.4: Thermostat operation: Thermostat classifies huge pages into hot and cold by randomly splitting a fraction of pages and estimating huge page access rate by poisoning a fraction of 4K pages within a huge page. The sampling policy is described in Section 6.3.2, Section 6.3.3 describes our approach to page access counting, and Section 6.3.4 describes the page classification policy. Note that the sampling and poisoning fractions here are for illustration purposes only. In our evaluation we sample 5% of huge pages and poison at most 50 4KB pages from a sampled huge page.

slow by design, the performance impact of this monitoring is low. In every sampling period we sort the huge pages in slow memory by their access counts and their aggregate access count is compared to the target access rate to slow memory. The most frequently accessed pages are then migrated back to fast memory until the access rate to the remaining cold pages is below the threshold. In addition to any mis-classified pages, this mechanism also identifies pages that become hot over time, adapting to changes in the application's hot working set.

6.3.6 Migration of cold pages to slow memory

Once cold pages have been identified by the guest, they must be migrated to the slow memory. We use the NUMA support in KVM guests to achieve this transfer. The NVM memory space is exposed to the guest OS as a separate NUMA zone, to which the guest OS can then transfer memory. NUMA support in KVM guests already exists in Linux and can be used via `libvirt` [90].

6.3.7 Thermostat example

Figure 6.4 illustrates Thermostat's page classification for an example application with eight huge pages. Each sampling period comprises three stages: (i) split a fraction of

huge pages, (ii) poison a fraction of split and accessed 4KB pages, record the access count to 4KB pages to estimate access rate of the huge pages, and (iii) classify pages as hot/cold. In this example, we sample 25% of huge pages (two huge pages out of eight are sampled). In the first sampling period, Thermostat splits and records 4KB-grain accesses to two pages (page 1 and 5) in the first scan period. In the second scan period, 4KB pages 1 and 4 of the first huge page and 3 and 8 of the fifth huge page are selected to be poisoned. Thermostat then estimates the access rate to huge page 1 and 5 from the access rates of the 4KB pages. Finally, Thermostat sorts the estimated access rates of the huge pages and classifies page 1 as cold, as its estimated access rate is below the threshold tolerable slow memory access rate. However, because the sum of the access rates of both huge pages is above the threshold access rate, page 5 is classified as hot. Similarly, in the second sampling period, pages 2 and 4 are randomly selected for sampling. At the end of the sampling period page 2 is classified as hot and page 4 as cold.

6.4 Methodology

6.4.1 System Configuration

We study Thermostat on a 36-core (72 hardware thread) dual-socket x86 server, Intel Xeon E5-2699 v3, with 512 GB RAM running Linux 4.5. Each socket has 45MB LLC. There is a 64-entry TLB per core and a shared 1024 entry L2 TLB. Several of our benchmark applications perform frequent I/O and are highly sensitive to OS page cache performance. To improve the page cache, we install `hugetmpfs`, a mechanism that enables use of huge pages for the `tmpfs` [91] filesystem. We place all files accessed by the benchmarks in `tmpfs`. In the future, we expect that Linux may natively support huge pages in the page cache for other file systems.

We run the benchmarks inside virtual machines using the Kernel-based Virtual Machine (KVM) virtualization platform. Client threads, which generate traffic to the servers, are run outside the virtual machine, on the host OS. We run the client threads and server VM on the same system and use a bridge network with virtio between host and guest so that network performance is not a bottleneck. We isolate the CPU and memory of the guest VM and client threads on separate sockets using Linux's control group mechanism [88] to avoid performance interference. The benchmark VM is allocated 8 CPU cores, a typical medium-sized cloud instance. We set the Linux frequency governor to "performance" to disable dynamic CPU frequency changes during application runs.

6.4.2 Emulating slow memory: BadgerTrap

Dual-technology main memory, in particular Intel/Micron's 3D XPoint memory, is not yet available. Hence, we use a software technique to emulate slow memory while placing all data in conventional DRAM.

Each cache miss to slow memory should incur an access latency that is a multiple of the DRAM latency (e.g., 400ns slow memory [76] vs. 50ns DRAM latency). There is no easy mechanism to trap to software on all cache misses. Instead, we introduce extra latency by inducing page faults upon translation misses (TLB misses) to cold pages by using BadgerTrap [89].

The software fault mechanism is an approximation of an actual slow memory device. The BadgerTrap fault latency (about 1us in our kernel) is higher than some authors predict the 3D XPoint memory read latency will be [76]. Furthermore, the poisoned PTE will induce a fault even if the accessed memory location is present in the hardware caches. In these two respects, our approach over-estimates the penalty of slow memory accesses. However, once BadgerTrap installs a (temporary) translation, further accesses to other cache blocks on the same slow-memory page will not induce additional faults, potentially under-estimating impact. Our testing with micro benchmarks indicates our approach yields an average access latency to slow memory in the desired range, in part, because slow-page accesses are sufficiently infrequent that they nearly always result in both cache and TLB misses anyway, as we discuss in Section 6.3.3.

One important detail of our test setup is that we must install BadgerTrap (for the purpose of emulating slow memory latency) within the guest VM rather than the host OS. Thermostat communicates with the guest-OS BadgerTrap instance to emulate migration to slow memory. We must install BadgerTrap within the guest because, otherwise, each BadgerTrap fault would result in a `vmexit`. In addition to drastically higher fault latency, `vmexit` operations have the side-effect of changing the Virtual Processor ID (VPID) to 0. Since KVM uses VPIDs to tag TLB entries of its guests, installing a TLB entry with the correct VPID would entail complexity and incur even higher emulation latency. Since BadgerTrap on the guest entails a latency of $\approx 1\mu$s, which is already higher than projected slow-memory latencies [76], we did not want to incur additional slowdown by emulating slow memory in the host OS.

6.4.3 Benchmarks

We evaluate Thermostat with applications from Google's Perfkit Benchmarker and the Cloudsuite benchmarks [92, 93]. These applications are representative server workloads

that have large memory footprints and are commonly run in virtualized cloud environments. We do not evaluate Thermostat for general-purpose GPU applications since non-volatile memory technologies are expected to have much lower bandwidth than high-bandwidth graphics memories – even lower than DDR memory technologies – thus are not a suitable match to memory bandwidth-sensitive platforms like GPUs.

TPCC on MySQL: TPCC is a widely-used database benchmark, which aims to measure the transaction processing throughput of a relational database [94]. We execute TPCC on top of MySQL, one of the most popular open-source database engines, which is often deployed in the cloud. We use the open-source TPCC implementation from OLTP-Bench [95] (available at `https://github.com/oltpbenchmark/oltpbench`). We use a scale factor of 320, and run the benchmark for 600 seconds after warming up for 600 seconds. MySQL makes frequent I/O requests and hence benefits markedly from our use of `hugetmpfs` to enable huge pages for the OS page cache.

NoSQL databases: Aerospike, Cassandra, and Redis are popular NoSQL databases [96, 97, 98]. Cassandra is a wide-column database designed to offer a variable number of fields (or columns) per key, while Redis and Aerospike are simpler key-value databases that have higher peak throughput. Redis is single-threaded whereas Aerospike is multi-threaded. Cassandra performs frequent file I/O as it periodically compacts its SSTable data structure on disk [99]. So, Cassandra also benefits substantially from `hugetmpfs`. Redis performs no file I/O after loading its dataset into memory.

We tune Aerospike, Cassandra, and Redis based on the settings provided by Google's Perfkit Benchkmarker for measuring cloud offerings [92]. We use the YCSB traffic generator to drive the NoSQL databases [80]. For Aerospike we use 200M operations and for Cassandra we use 50M operations on 5M keys with 20 fields each with a Zipfian distribution. For both of these application, we evaluate two workload mixes: a read-heavy load with 95:5 read/write ratio and a write-heavy load with 5:95 read/write ratio. For Redis, we access keys with a hotspot distribution, wherein 0.01% of the keys account for 90% of the traffic. We vary value sizes according to the distribution reported in [100]. We observe 176K and 215K operations/sec for read-heavy and write-heavy workloads for Aerospike, and 21K and 45K operations/sec for read-heavy and write-heavy workloads for Cassandra. For Redis we observe 188K operations/sec for our baseline system with all pages in DRAM as huge pages.

In-memory analytics: We evaluate Thermostat on in-memory analytics benchmarks from Cloudsuite [93]. In-memory analytics runs a collaborative filtering algorithm on a dataset of user-movie ratings. It uses the Apache Spark framework to perform data analytics. We set both executor and driver memory to be 6GB to execute the benchmark

	Resident Set Size	File-mapped
Aerospike	12.3GB	5MB
Cassandra	8GB	4GB
MySQL-TPCC	6GB	3.5GB
Redis	17.2GB	1MB
In-memory-analytics	6.2GB	1MB
Web-search	2.28GB	86MB

Table 6.2: Application memory footprints: resident set size and file-mapped pages.

entirely in memory. We run the benchmark to completion, which takes 317 seconds for our baseline system with all pages in DRAM as huge pages.

Web search: Cloudsuite's web search uses the Apache Solr search engine framework. We run client threads on host and index nodes within the virtual machine. We set steady state time to be 300 seconds and keep default values for all the other parameters on the client machine. As specified by the benchmark, target response time requires 99% of the requests to be serviced in 200ms. For our baseline system with all pages in DRAM as huge pages, we observe 50 operations/sec with $\approx$ 85ms 99th percentile latency.

6.4.4 Runtime overhead of Thermostat Sampling

We measure the runtime overhead of Thermostat to ensure that application throughput is not degraded by Thermostat's page sampling mechanism. For sampling periods of 10s or higher, we observe negligible CPU activity from Thermostat and no measurable application slowdown ($<$ 1%).

6.5 Evaluation

We next measure Thermostat's automatic hot/cold classification and run-time placement/migration mechanisms on our suite of cloud applications.

Thermostat takes as input a tolerable slowdown; a single input parameter specified by a system administrator. It then automatically selects cold pages to be migrated to slow memory at runtime. We set a tolerable slowdown of 3% throughout our evaluation, since a higher slowdown may lead to an overall cost increase due to higher required CPU provisioning (which is more expensive than memory). Thermostat's slowdown threshold can be changed at runtime through the Linux cgroup mechanism. Hence, application administrators can dynamically tune the threshold based on service level agreements for

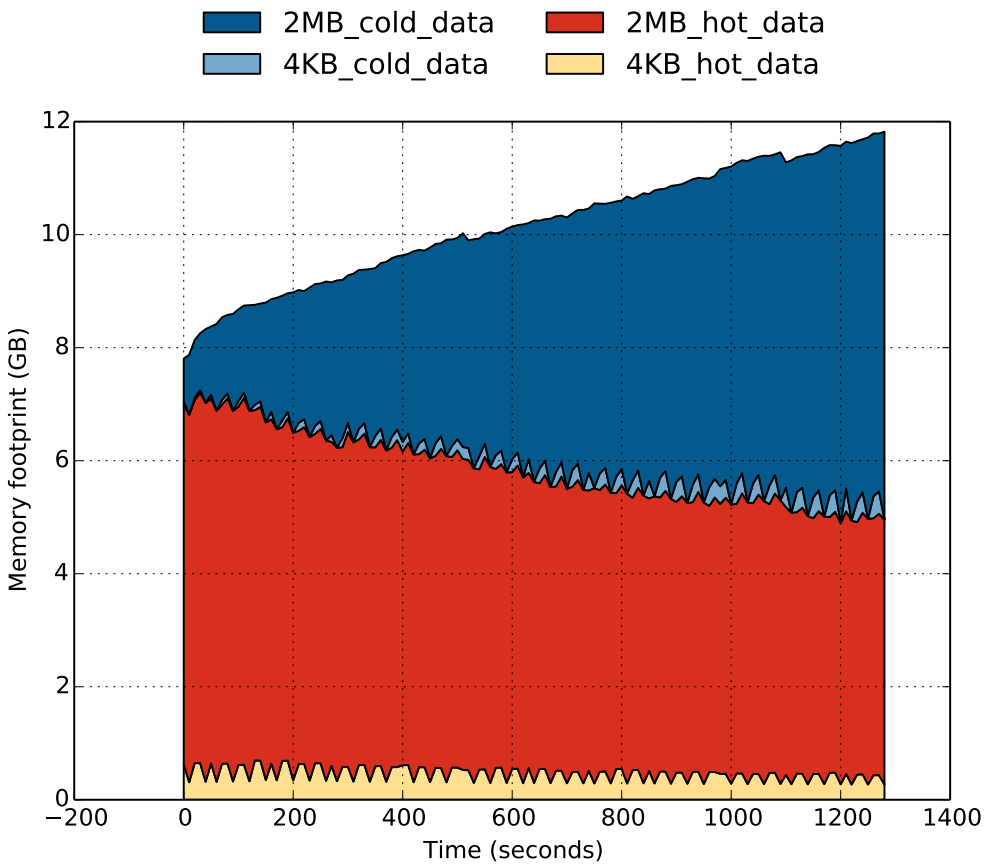


Figure 6.5: Amount of cold data in Cassandra identified at run time with 2% throughput degradation for a write-heavy workload (5:95 read/write ratio).

latency-critical applications or for the throughput requirements of batch jobs. We show that, for our application suite, Thermostat meets the target 3% slowdown while placing a significant fraction of application footprint in slow memory dynamically at runtime.

We evaluate Thermostat with 5% of huge pages sampled in every scan interval of 30s and at most 50 4KB pages poisoned for a sampled huge page. We compare the performance of Thermostat with a placement policy that places all pages in DRAM, which maximizes performance while incurring maximal memory cost. Thermostat's sampling mechanisms incur a negligible performance impact (well under 1%)—the slowdowns we report are entirely attributable to slow memory accesses.

We briefly discuss our findings for each application. Table 6.2 reports each application's footprint in terms of resident set size (RSS) and file-mapped pages. The memory savings quoted for each benchmark is the average cold memory fraction over the benchmark's runtime.

Cassandra: We report the breakdown of hot and cold 2MB and 4KB pages over time

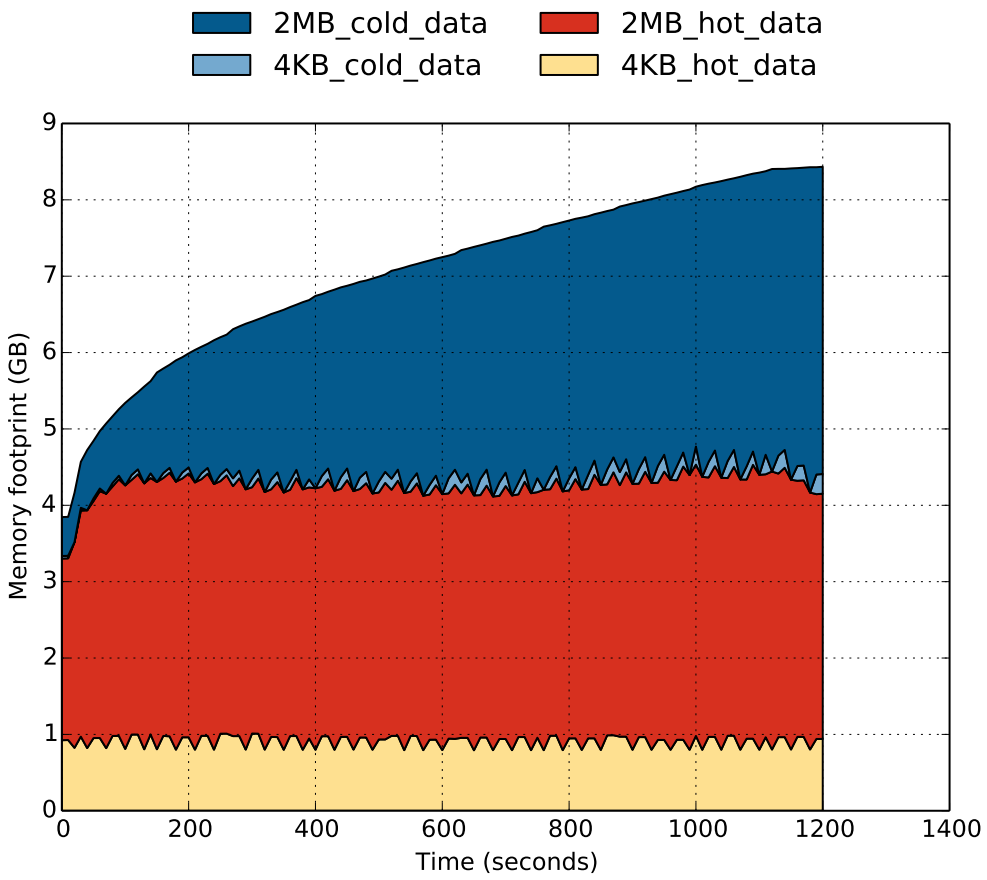


Figure 6.6: Amount of cold data in MySQL-TPCC identified at run time with 1.3% throughput degradation.

for Cassandra in Figure 6.5 for the write-heavy workload. Thermostat identifies between 40-50% of Cassandra's footprint (including both 4KB and 2MB pages) as cold. Note that the cold 4KB pages are solely due to splitting of cold huge pages during profiling ($\approx$ 5% of cold pages are 4KB, since our profiling strategy is agnostic of a page being hot or cold). Note that the resulting throughput degradation of 2% falls slightly under our target of 3%. We observe $\approx$ 1% higher average, 95th, and 99th percentile read/write latency for Cassandra with Thermostat. Based on this performance vs. footprint trade-off, we estimate Thermostat enables a net cost savings of $\approx$ 30% for Cassandra (see Section 6.5.3 for a detailed analysis). For the read-heavy workload Thermostat identifies 40% of data as cold with 2.5% throughput degradation (we omit figure due to space constraints).

The memory consumption of Cassandra grows due to in-memory Memtables filling up. The Memtable is flushed to disk in the form of an SSTable, which then leads to a sharp decrease in memory consumption. However, we do not observe such a compaction event

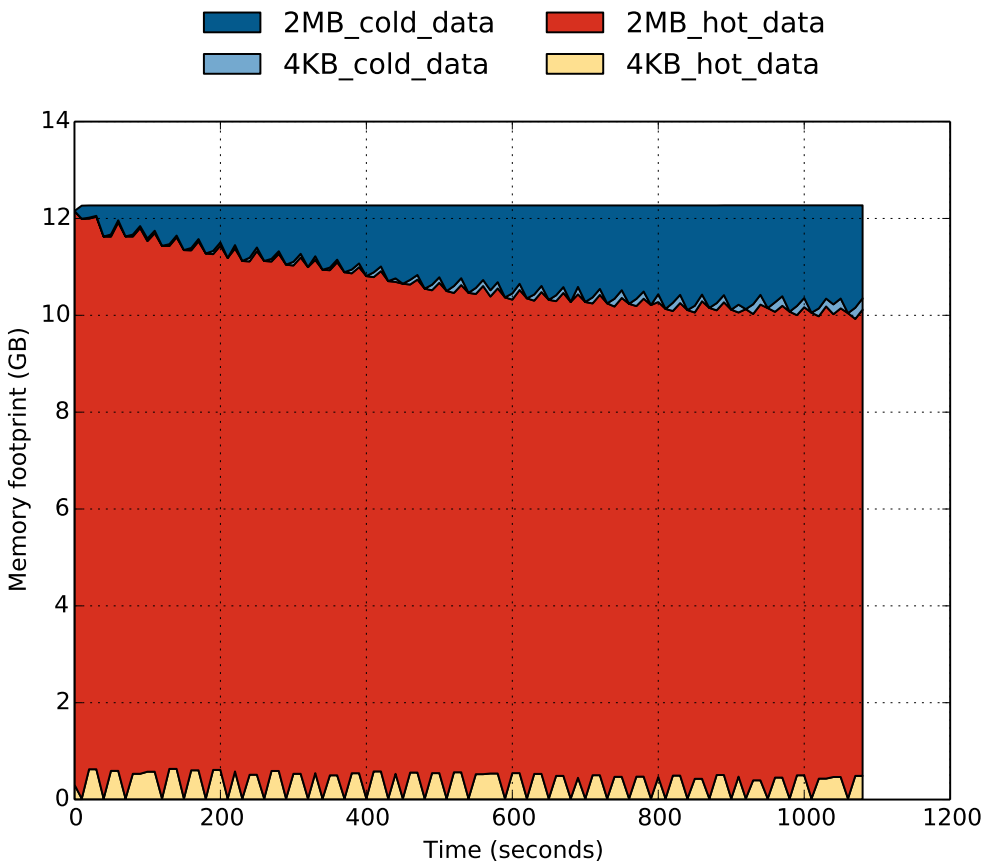


Figure 6.7: Amount of cold data in Aerospike identified at run time with 1% throughput degradation for a read-heavy workload (95:5 read/write ratio).

in our configuration, due to the large amount of memory provisioned for Cassandra in our test scenario.

MySQL-TPCC: In Figure 6.6 we show a similar footprint graph for MySQL-TPCC. The largest table in the TPCC schema, the LINEITEM table, is infrequently read. As a result, much of TPCCs footprint (about 40-50%) is cold and can be placed in slow memory while limiting performance degradation to 1.3%.

Aerospike: In Figure 6.7 we show a similar footprint graph for Aerospike for the read-heavy workload. We see a small fraction of the footprint (about 15%) identified as cold while maintaining the tolerable slowdown. The average, 95th and 99th read/write latencies are all within 3% of the baseline. For the write-heavy workload, Thermostat identifies about 15% of data as cold while satisfying tolerable slowdown (we omit figure due to space constraints).

Redis: Unlike the other applications, Redis has a more uniform access pattern. The key data structure in Redis is a large hash table, hence, memory accesses are spread

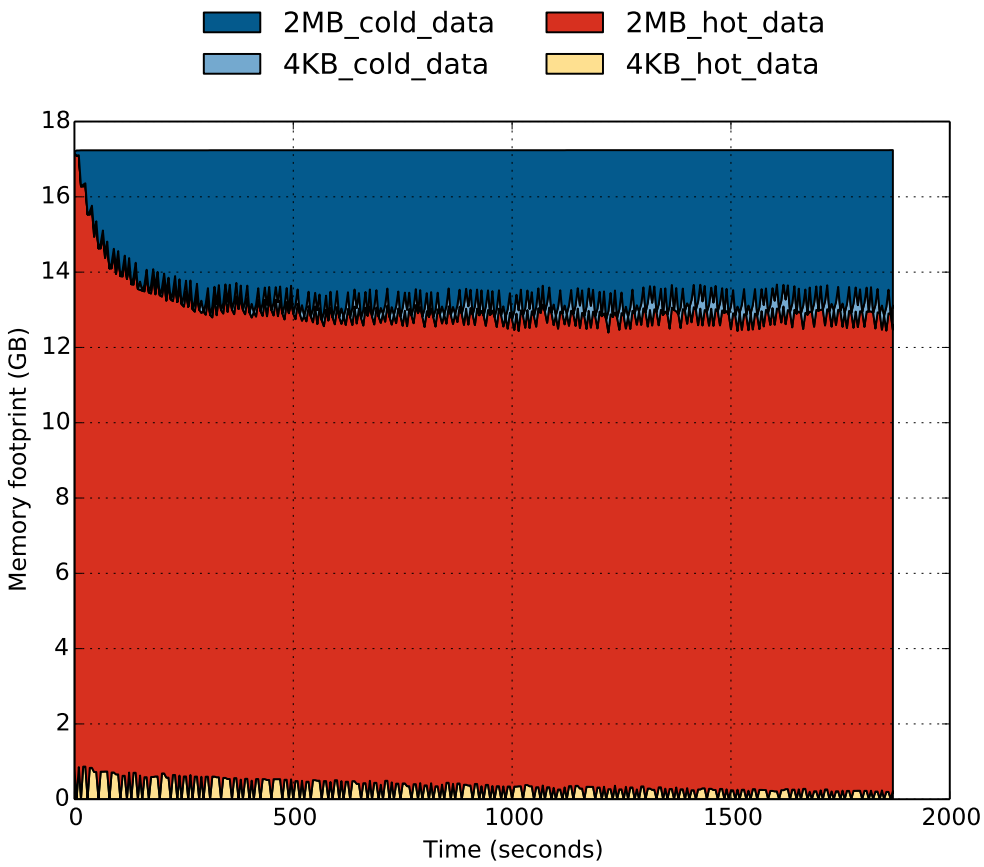


Figure 6.8: Amount of cold data in Redis identified at run time with 2% throughput degradation.

relatively uniformly over its address space; the relative hotness of pages reflects the corresponding hotness of the key distribution. We study a load where 0.01% of keys account for 90% of accesses. In Figure 6.8 we show that, under this load, 10% of the data is detected as cold at a 3% throughput degradation. The average read/write latency is 3.5% higher than the baseline.

In-memory analytics: We also evaluate in-memory analytics benchmark from Cloudsuite [93]. In Figure 6.9 we show Thermostat detects about 15-20% data as cold. As application footprint grows, Thermostat scans more pages and thus the cold page fraction also grows with time. We run this benchmark to completion, however, the benchmark runtime is much shorter than the previous data serving applications. (Cloudsuite is designed for tractable runtimes under expensive instrumentation and/or simulation). Nevertheless, Thermostat successfully identifies cold data while meeting the slowdown target. We expect the cold memory footprint of this application to reach steady state if a larger input were available.

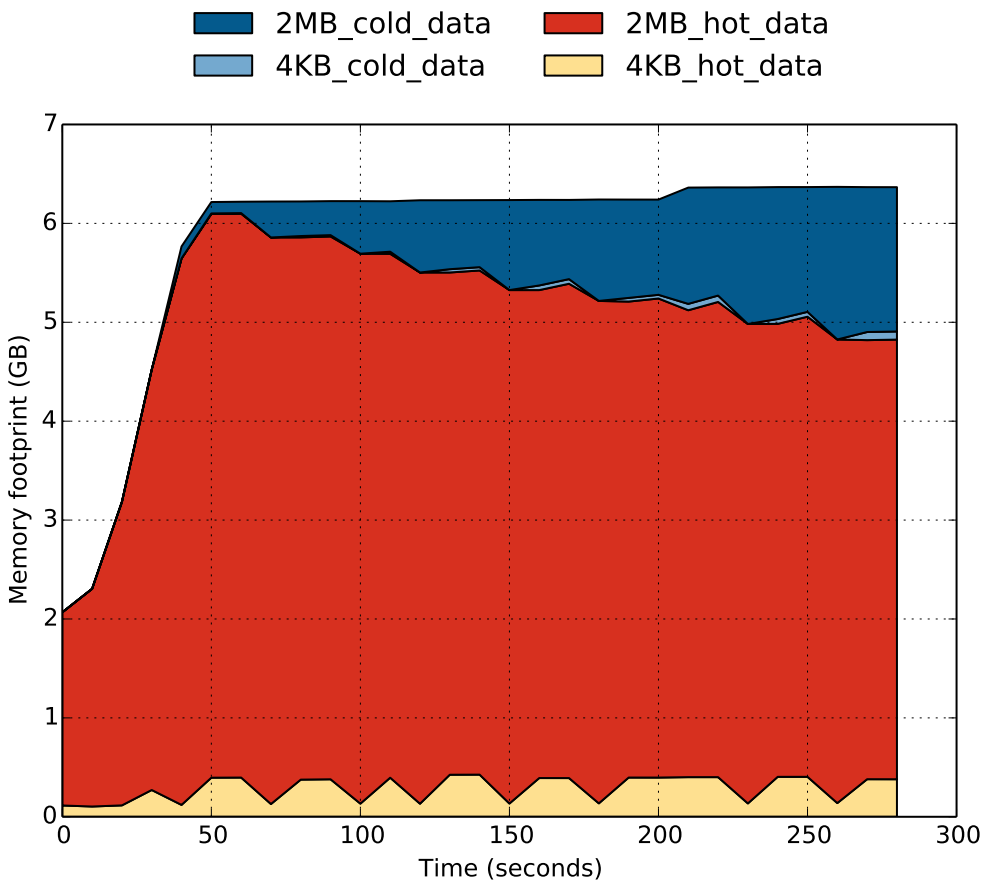


Figure 6.9: Amount of cold data in in-memory analytics benchmark identified at run time with 3% runtime overhead.

Web search: In Figure 6.10 we show the footprint graph for the web search workload. We see about 40% of the footprint identified as cold. We observe $<$ 1% degradation in throughput and no observable degradation in 99th percentile latency of 200ms.

6.5.1 Sensitivity to tolerable slowdown target

Next we show the sensitivity of Thermostat to the single input parameter specified by a system administrator, the tolerable slowdown. In our baseline evaluation, we set this parameter to 3%. However, due to changes in the price point of the memory technology or changes in data-center cost structure it may be possible to tolerate higher slowdown. To study the adaptability of Thermostat in such scenarios we also evaluate 6% and 10% slowdown targets. We show the variation in amount of cold data identified by Thermostat at run time with tolerable slowdown in Figure 6.11.

We observe that with increase in tolerable slowdown Thermostat can place a higher

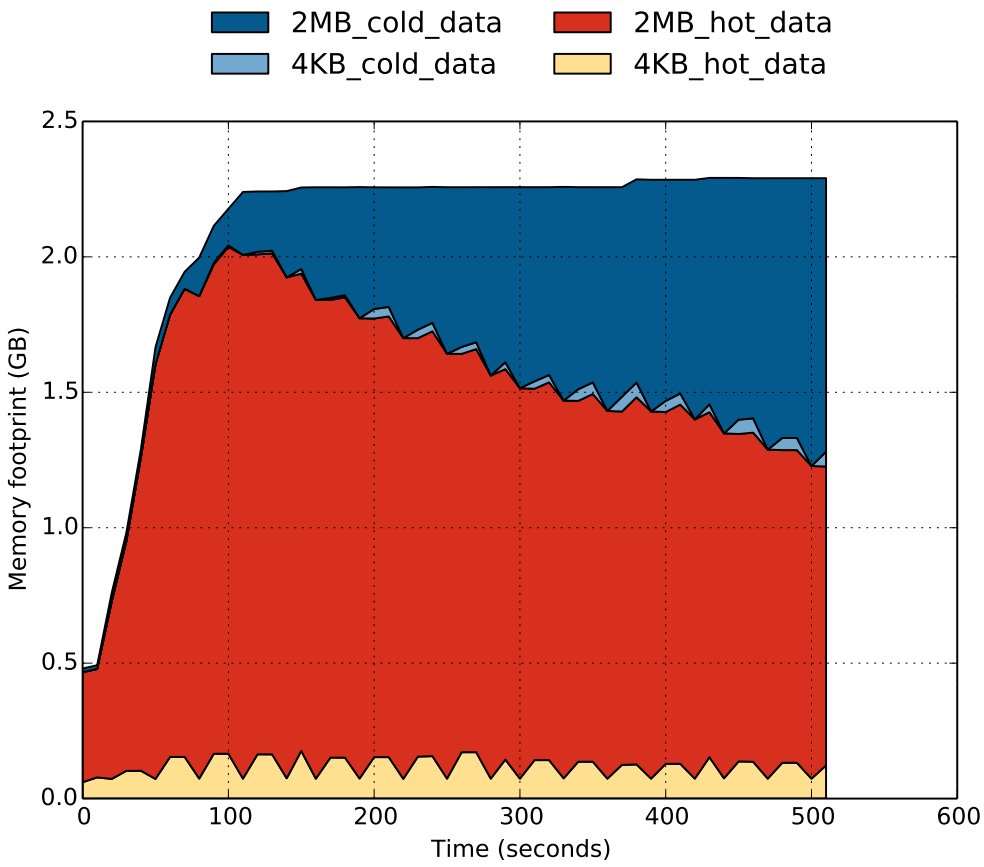


Figure 6.10: Amount of cold data in web-search benchmark identified at run time with 1% throughput and no 99th percentile latency degradation.

fraction of memory footprint in slow memory. We also observe that the performance targets of all applications are met (we omit data due to space constraints). However, in several cases, the achieved slowdown is less than the specified slowdown.

For MySQL-TPCC, Thermostat is not able to identify additional cold data even with an increase in tolerable slowdown (cold data fraction saturates at $approx$ 45%). This saturation happens because all remaining pages for TPCC are highly accessed, and placing any of them in slow memory results in an unacceptable application slowdown. For Aerospike, Thermostat is able to scale the cold data with varying tolerable slowdown. However, the actual slowdown doesn't reach the target slowdown due to (a) Aerospike's performance being insensitive to cold page accesses, and (b) the average OS fault handler latency for emulation being lower than the assumed latency of 1us used in Thermostat.

In summary, Thermostat places a higher fraction of data in slow memory if the user can tolerate more slowdown. This feature allows system administrators to better utilize expensive DRAM capacity by moving as much cold data to slow memory as possible via

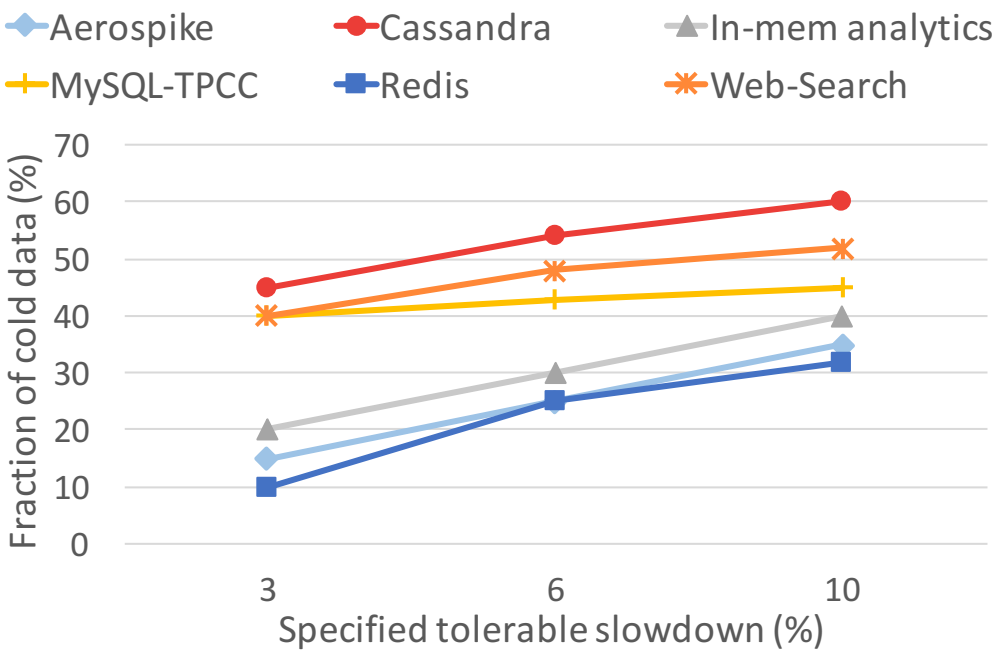


Figure 6.11: Amount of cold data in identified at run time varying with the specified tolerable slowdown. All the benchmarks meet their performance targets (not shown in the figure) while placing cold data in the slow memory.

Thermostat.

6.5.2 Migration overhead and slow memory access rate

To verify that Thermostat doesn't cause un-realizable bandwidth pressure on the slow memory, we measured the memory bandwidth required by migrations and false classifications between slow and fast memory. In Table 6.3 we observe that the required migration bandwidth is $<$ 30 MB/s on average across all benchmarks. The highest total traffic to cold memory we observe is 60 MB/s, which is well within the projected capability of near-

(MB/s)	Migration	False-classification
Aerospike	13.3	9.2
Cassandra	9.6	3.8
In-mem-Analytics	16	0.4
MySQL-TPCC	6	1.8
Redis	11.3	10
Web-search	1.6	0.3

Table 6.3: Data migration rate and false classification rate of slow memory. These rates are well-below expected bandwidth of near-future memory technologies.

Slow memory cost relative to DRAM	$0.33\times$	$0.25\times$	$0.2\times$
Aerospike	10%	11%	12%
Cassandra	27%	30%	32%
In-mem-Analytics	11%	12%	13%
MySQL-TPCC	27%	30%	32%
Redis	17%	19%	20%
Web-search	27%	30%	32%

Table 6.4: Memory spending savings relative to an all-DRAM system when using slow memory with different cost points relative to DRAM.

future cheap memory technologies [76]. Thus, we infer that Thermostat doesn't produce unrealistic pressure on the memory system.

6.5.3 DRAM Cost Analysis

Since DRAM pricing is volatile, and slow memory prices remain unclear, it is difficult to perform a rigorous cost-savings analysis for Thermostat. We use a simple model to estimate the cost-savings possible with Thermostat and study a range of possible ratios of DRAM to slow-memory cost. Table 6.4 shows the fraction of DRAM spending saved due to Thermostat when slow memory is $\frac{1}{3}$, $\frac{1}{4}$ and $\frac{1}{5}$ of DRAM cost. We can see that, depending on workload and memory technology, anywhere from $\approx$ 10% (for Aerospike) to 32% (for Cassandra) of DRAM cost can be saved.

6.6 Discussion

6.6.1 Hardware support for access counting

The software-only page access counting mechanism described in Section 6.3.3 is desirable since it can be run on *current* commodity x86 hardware. However it has two sources of inaccuracy: (i) we can only count TLB misses instead of LLC misses, and (ii) the measurement process itself throttles accesses to the poisoned pages, since the poison faults to the same page are serialized. We describe below two extensions to x86 hardware that can address these shortcomings.

6.6.1.1 Fault on LLC miss:

To accurately count the number of actual cache misses caused by a page, the x86 PTE can be modified to include a "count miss" (CM) bit. The CM bit should be stored in the corresponding TLB entry as well. When the CM bit is set in a translation entry, any LLC miss to that page will result in a software fault. The CM fault handler can then be used to track the number of cache misses to that page in the same way that BadgerTrap uses the reserved bit fault handler to track number of TLB misses. The instruction triggering a CM fault should retire once the data is fetched, since replaying the instruction could lead to a cascade of CM faults. During a CM fault, the actual memory (DRAM or NVRAM) access can be done in parallel with servicing the fault to partially or fully hide the CM fault service latency.

6.6.1.2 PEBS based access counting:

In this scheme, the PEBS (Precise Event Based Sampling) subsystem in the x86 architecture [84] can be extended to record page access information. In the current PEBS implementation, a PEBS record is stored in a pre-defined memory area on samples of specific events (LLC misses is one of them). When the area fills up, an interrupt is raised, which the kernel can then service and inspect the instructions that led to those events.

The maximum PEBS sampling frequency is limited by how fast the memory area can be filled and kernel interrupts serviced. The default value in the Linux kernel for PEBS sampling frequency is 1000Hz, which is far too low to support $\approx$ 30,000 slow memory accesses that can be done by a single thread for a 3% performance slowdown. If the record entry *only* stores the physical page address of the access, it can be stored in 48b, which is far less than the entire CPU state.

Merits to slow memory software-emulation: One of the major challenges in deploying new hardware in data centers is to evaluate impact on throughput degradation and tail latency to avoid violating service level agreements. Thermostat can be used in test nodes of production systems today to evaluate the performance implication of deploying slow memory in data centers. Thermostat does not need any specialized test hardware and is pluggable with a parameterized delay for simulating slow memory. Approaches without extensive hardware changes are likely to receive more widespread adoption in the industry [23]. Thus, using our evaluation system, we can easily evaluate the impact of using slow memory in an application for a given traffic pattern. For example, we experimented with a Zipfian traffic pattern for Redis and failed to place more that 10% of its footprint in slow memory without significant throughput degradation. Thus, such a tool allows one to

evaluate the potential usability of slow memory in production *without any extra software instrumentation or hardware investment*.

Device wear: Some likely candidates for cheap, slow memory technologies are subject to device wear, which means that the memory may not function properly if it is written too frequently. Qureshi et al. propose a simple wear-leveling technique that uses an algebraic mapping between logical addresses and physical addresses along with address-randomization techniques to improve the lifetime of memory devices subject to wear [101]. Table 6.3 shows that accesses to slow memory by Thermostat fall well below the expected endurance limits of future memory technologies.

Spreading a 2MB page across fast and slow memories: In our scheme, the entirety of a 2MB is placed in slow or fast memory. This has the benefit of reducing TLB misses, but consumes extra fast memory space for 2MB pages with a small hot footprint. The evaluation of a scheme which selectively places only hot portions of an otherwise cold 2MB page in fast memory is left for future work.

6.7 Conclusion

With recent announcements of 3D XPoint memory by Intel/Micron there is an opportunity of cutting cost of memories in data centers with improved capacity and cost per bit. However, due to projected higher access latencies of these new memory technologies it is not feasible to completely replace main memory DRAM technology. To address the renewed interest in two-tiered physical memory we presented and evaluated Thermostat, an application-transparent huge-page-aware mechanism to place pages in a dual technology hybrid memory system, while achieving both the cost advantages of two-tiered memory and performance advantages of transparent huge pages. Huge pages, being performance critical in cloud applications with large memory footprints, especially in virtualized cloud environments, need to be supported in this two-tier memory system. We present a new hot/cold classification mechanism to distinguish frequently accessed pages (hot) from infrequently accessed ones (cold). We implement Thermostat in Linux kernel version 4.5 and show that it can transparently move cold data to slow memory while satisfying a 3% tolerable slowdown. We show that our online cold page identification mechanism incurs no observable performance overhead and can migrate up to 50% of application footprint to slow memory while limiting performance degradation to 3%, thereby reducing memory cost up to 30%.

CHAPTER VII

Related Work

7.1 Page Placement And Migration

Using mixed DRAM technologies or DRAM in conjunction with non-volatile memories to improve power consumption on CPUs has been explored by several groups [102, 103, 104, 105, 106]. The majority of this work attempts to overcome the performance reductions introduced by non-DDR technologies to improve capacity, power consumption, or both. In CC-NUMA systems, there has been a long tradition of examining where to place memory pages and processes for optimal performance, typically focusing on reducing memory latency [107, 108, 109, 110, 111, 112]. Whereas CPUs are highly sensitive to memory latency, GPUs can cover a much larger latency through the use of multi-threading. More recent work on page placement and migration [79, 60, 113, 114, 115, 116, 117] has considered data sharing characteristics, interconnect utilization, and memory controller queuing delays in the context of CPU page placement. However, the primary improvements in many of these works, reducing average memory latency, will not directly apply in a GPU optimized memory system.

Several recent papers have explored hybrid DRAM-NVM GPU attached memory subsystems [118, 119]. Both of these works consider a traditional GPU model where the availability of low latency, high bandwidth access to CPU-attached memory is not considered, nor are the overheads of moving data from the host CPU onto the GPU considered. Several papers propose using a limited capacity, high bandwidth memory as a cache for a larger slower memory [120, 121], but such designs incur a high engineering overhead and would require close collaboration between GPU and CPU vendors that often do not have identically aligned visions of future computing systems.

When designing page migration policies, the impact of TLB shootdown overheads and page table updates is a constant issue. Though most details about GPU TLBs are not

public, several recent papers have provided proposals about how to efficiently implement general purpose TLBs that are, or could be, optimized for a GPU's needs [56, 55, 57]. Others have recently looked at improving TLB reach by exploiting locality within the virtual to physical memory remapping, or avoiding this layer completely [122, 123, 86]. Finally, Gerofi et al. [124] recently examined TLB performance of the Xeon Phi for applications with large footprints, while McCurdy et al. [125] investigated the effect of superpages and TLB coverage for HPC applications in the context of CPUs.

7.2 Cache Coherence

Cache coherence for CPUs has received great attention in the literature. Recent proposals have started to explore intra-GPU and CPU–GPU cache coherence.

CPU Systems: Scalable cache coherence has been studied extensively for CPU-based multicore systems. Kelm et al. show that scaling up coherence to hundreds or thousands of cores will be difficult without moving away from pure hardware-based coherence [126, 127], due to high directory storage overheads and coherence traffic [70, 128]. Whereas some groups have evaluated software shared memory implementations [129, 127], Martin et al. argue that hardware cache coherence for mainstream processors is here to stay, because shifting away from it simply shifts the burden of correctness into software instead of hardware [39]. Nevertheless, disciplined programming models coupled with efficient hardware implementations are still being pursued [130, 131, 132].

Self-invalidation protocols have been proposed to reduce invalidation traffic and reduce coherence miss latency [70, 71]. Our cacheless request coalescing scheme uses a similar idea, discarding a block immediately after fulfilling requests pending at the MSHR. Other proposals have classified data into private, shared, and instruction pages and have devised techniques to curtail coherence transactions for private data [40, 133, 134, 135]. We instead classify pages into read-only versus read-write and exploit the fact that read-only data can be safely cached in incoherent caches.

Ros and Kaxiras [135] have proposed a directory-less/broadcast-less coherence protocol where all shared data is self-invalidated at synchronization points. In this scheme, at each synchronization point (e.g., lock acquire/release, memory barrier) all caches need to be searched for shared lines and those lines have to be flushed—an expensive operation to implement across hundreds of GPU caches with data shared across thousands of concurrent threads.

Heterogeneous Systems and GPUs: With the widespread adoption of GPUs as a primary computing platform, the integration of CPU and GPU systems has resulted in

multiple works assuming that CPUs and GPUs will eventually become hardware cache-coherent with shared page tables [57, 56, 14, 13]. CPU–GPU coherence mechanisms have been investigated, revisiting many ideas from distributed shared memory and coherence verification [136, 22, 137, 138]. Power et al. [22] target a hardware cache-coherent CPU–GPU system by exploiting the idea of region coherence [41, 139, 140, 141]. They treat the CPU and the GPU as separate regions and mitigate the effects of coherence traffic by replacing a standard directory with a region directory. In contrast, we identify that CPUs and GPUs need not be cache-coherent; the benefits of unified shared memory with correctness guarantees can also be achieved via selective caching, which has lower implementation complexity.

Mixing incoherent and coherent shared address spaces has been explored before in the context of CPU-only systems [142] and the appropriate memory model for mixed CPU–GPU systems is still up for debate [38, 21, 143, 144]. Hechtman et al. propose a consistency model for GPUs based on release consistency, which allows coherence to be enforced only at release operations. They propose a write-through no-write-allocate write-combining cache that tracks dirty data at byte granularity. Writes must be flushed (invalidating other cached copies) only at release operations. Under such a consistency model, our selective caching scheme can be used to avoid the need to implement hardware support for these invalidations between the CPU and GPU.

Cache coherence for GPU-only systems has been studied by Singh et al. [145], where they propose a timestamp-based hardware cache-coherence protocol to self-invalidate cache lines. Their scheme targets single-chip systems and would require synchronized timers across multiple chips when implemented in multi-chip CPU-GPU environments. Kumar et al. [146] examine CPUs and fixed-function accelerator coherence, balancing coherence and DMA transfers to prevent data ping-pong. Suh et al. [147] propose integrating different coherence protocols in separate domains (such as MESI in one domain and MEI in another). However, this approach requires invasive changes to the coherence protocols implemented in both domains and requires significant implementation effort by both CPU and GPU vendors.

Bloom Filters: Bloom Filters [148] and Cuckoo Filters [149, 68] have been used by several architects [150, 151, 152] in the past. Fusion coherence [137] uses a cuckoo directory to optimize for power and area in a CMP system. JETTY filters [153] have been proposed for reducing the energy spent on snoops in an SMP system. We use a cuckoo filter to implement the GPU remote directory.

7.3 Two-level Memory

Application-guided two-level memory: Dulloor et al. [76] propose X-Mem, a profiling based technique to identify data structures in applications that can be placed in NVM. To use X-Mem, the application has to be modified to use special `xmalloc` and `xfree` calls instead of `malloc` and `free` and annotate the data structures using these calls. Subsequently, the X-Mem profiler collects a memory access trace of the application via PinTool [154], which is then post-processed to obtain access frequencies for each data structure. Using this information, the X-Mem system places each data structure in either fast or slow memory.

Such an approach works well when: (1) an overwhelming majority of the application memory consumption is allocated by the application itself, (2) modifiable application source code is available along with a deep knowledge of the application data structures, and (3) a representative profile run of an application can be performed. Most cloud-computing software violates some, or all, of these criteria. As we show in Section 6.5, NoSQL databases interact heavily with the OS page cache, which accounts for a significant fraction of their memory consumption. Source code for cloud applications is not always available. Moreover, even when source code is available, there is often significant variation in hotness within data structures, e.g., due to hot keys in a key-value store. Obtaining representative profile runs is difficult due to high variability in application usage patterns [100]. In contrast, Thermostat is application transparent, can dynamically identify hot and cold regions in applications at a page granularity (as opposed to data structure granularity), and can be deployed seamlessly in multi-tenant host systems.

Lin et al. present a user-level API for memory migration and an OS service to perform asynchronous, hardware-accelerated memory moves[155]. However, Thermostat is able to avoid excessive page migrations by accurately profiling and identifying hot and cold pages, and hence can rely on Linux's existing page migration mechanism without suffering undue throughput degradation. Agarwal et al. [14, 13] demonstrate the throughput advantages of profile guided data placement for GPU applications. Chen et al. [156] present a portable data placement engine directed towards GPUs. Both of these proposals seek to maximize bandwidth utilization across memory sub-systems with disparate peak bandwidth capability. They focus on bandwidth- rather than latency-sensitive applications and do not seek to reduce system cost.

Linux provides the `madvise` API for applications to provide hints about application memory usage. Jang et al. propose an abstraction for *page coloring* to enable communication between applications and the OS to indicate which physical page should be

used to back a particular virtual page. Our approach, however, is application transparent and does not rely on hints, eliminating the programmer burden of rewriting applications to exploit dual-technology memory systems.

Software-managed two-level memory: AutoNUMA [79] is an automatic placement/migration mechanism for co-locating processes and their memory on the same NUMA node to optimize memory access latency. AutoNUMA relies on CPU-page access affinity to make placement decisions. In contrast, Thermostat makes its decisions based on the page access rate, irrespective of which CPU issued the accesses.

Hardware-managed two-level memory: Several researchers [157, 77] have proposed hybrid dual technology memory organizations with hardware enhancements. In such approaches, DRAM is typically used as a transparent cache for a slower memory technology. Such designs require extensive changes to the hardware memory hierarchy; Thermostat can place comparable fractions of the application footprint in slower memory without any special hardware support in the processor or caches.

Disaggregated memory: Lim et al. [36, 38] propose a disaggregated memory architecture in which a central pool of memory is accessed by several nodes over a fast network. This approach reduces DRAM provisioning requirements significantly. However, due to the high latency of network links, performing remote accesses at cache-line level granularity is not fruitful, leading Lim to advocate a remote paging interface. For the latency range that Lim assumed (12-15us), our experience confirms that application slowdowns are prohibitive. However, for the expected sub-microsecond access latency of near-future cheap memory technologies, our work shows that direct cache-line-grain access to slow memory can lead to substantial cost savings.

Cold data detection: Cold/stale data detection has been extensively studied in the past, mainly in the context of paging policies and disk page caching policies, including mechanisms like `kstaled` [81, 82, 158]. Our work differs from such efforts in that we study the impact of huge pages on cold data detection, and show a low-overhead method to detect and place cold data in slow memory without significant throughput degradation. Baskakov et al. [159] and Guo et al. [83] propose a cold data detection scheme by breaking 2MB pages into 4KB pages so as to detect cold-spots in large 2MB pages and utilizing PTE "Accessed" bits. However, as we have shown in Section 6.2.1, obtaining performance degradation estimates from Accessed bits is difficult. Thus, we instead use finer grain access frequency information obtained through page poisoning.

NVM/Disks: Several upcoming memory technologies are non-volatile, as well as slower and cheaper than DRAM. The non-volatility of such devices has been exploited by works like PMFS [160] and pVM [161]. These works shift a significant fraction of persis-

tent data from disks to non-volatile memories (NVMs), and obtain significant performance benefits due to the much faster speed of NVMs as compared to disks. Our work explores placing volatile data in cheaper memory to reduce cost, and as such is complementary to such approaches.

Hardware modifications for performance modeling: Li et al. [78] propose a set of hardware modifications for gauging the impact of moving a given page to DRAM from NVM based on its access frequency, row buffer locality and memory level parallelism. The most impactful pages are then moved to DRAM. In a similar vein, Azimi et al. [162] propose hardware structures to gather LRU stack and miss rate curve information online. However, unlike our work, these techniques require several modifications to the processor and memory controller architecture. Existing performance counter and PEBS support has been utilized by prior works [163, 164] to gather information related to the memory subsystem. However, gathering page granularity access information at higher frequency from the PEBS subsystem requires a high overhead and so is not desirable.

CHAPTER VIII

Conclusion

Current OS page placement policies are optimized for both homogeneous memory and latency sensitive systems. With the emergence of shared virtual address CPU-GPU systems and new memory technologies like Intel/Micron's 3D XPoint memory management policies need to account for difference in bandwidth, coherence domains, cost per dollar of different memory technologies, deployed to use concurrently by the computing agents.

The first problem we examine is one that the GPU industry is facing on how to best handle memory placement for upcoming cache coherent GPU-CPU systems. While the problem of page placement in heterogeneous memories has been studied extensively in the context of CPU-only systems, the integration of GPUs and CPUs provides several unique challenges. First, GPUs are extremely sensitive to memory bandwidth, whereas traditional memory placement decisions for CPU-only systems have tried to optimize latency as their first-order concern. Second, while traditional SMP workloads have the option to migrate the executing computation between identical CPUs, mixed GPU-CPU workloads do not generally have that option since the workloads (and programming models) typically dictate the type of core on which to run. Finally, to support increasingly general purpose programming models, where the data the GPU shares a common address space with the CPU and is not necessarily known before the GPU kernel launch, programmer-specified up-front data migration is unlikely to be a viable solution in the future.

We propose a new BW-AWARE page placement policy that uses memory system information about heterogeneous CPU-GPU memory system characteristics to place data appropriately, achieving 35% performance improvement on average over existing policies without requiring any application awareness. We also present an intelligent dynamic page migration solution that maximizes the bandwidth utilization of different memory technologies in heterogeneous CPU-GPU memory system. We identify that demand-based

migration alone is unlikely to be a viable solution due to both application variability and the need for aggressive prefetching of pages the GPU is likely to touch, but has not touched yet. The use of range expansion based on virtual address space locality, rather than physical page counters, provides a simple method for exposing application locality while eliminating the need for hardware counters exploiting the memory access pattern of GPU computer applications. Developing a system with minimal hardware support is important in the context of upcoming CPU-GPU systems, where multiple vendors may be supplying components in such a system and relying on specific hardware support on either the GPU or CPU to achieve performant page migration may not be feasible.

Introducing globally visible shared memory in future CPU/GPU systems improves programmer productivity and significantly reduces the barrier to entry of using such systems for many applications. Hardware cache coherence can provide such shared memory and extend the benefits of on-chip caching to all memory within the system. However, extending hardware cache coherence throughout the GPU places enormous scalability demands on the coherence implementation. Moreover, integrating discrete processors, possibly designed by distinct vendors, into a single coherence protocol is a prohibitive engineering and verification challenge.

In this thesis we also explore the problem of CPU-GPU coherence. Despite its programmability benefits, cache coherence between CPU and GPU can be a daunting design challenge due to coordination required between different vendors to implement such a solution. To mitigate this problem, we propose Selective Caching, a technique that disallows GPU caching of memory touched by CPU so as to maintain coherence without requiring cache coherence. We show that Selective Caching coupled with request coalescing, a CPU side GPU client cache, and variable sized transfer units can reach 93% of the performance of a cache-coherent system.

The second major avenue we look at is the usage of cheaper (per-bit) but slower memory technologies in cloud deployments. Upcoming technologies such as Intel/Micron's 3D-XPoint can significantly reduce data-center costs by reducing provisioning of costly DRAM. However, the higher latencies of these memory technologies mean that they can not replace DRAM fully. Thus, heterogeneous memory systems with both slow and fast memory interfaces are likely to be deployed by cloud vendors. To address the renewed interest in two-tiered physical memory we present and evaluate Thermostat, an application-transparent huge-page-aware mechanism to place pages in a dual technology hybrid memory system, while achieving both the cost advantages of two-tiered memory and performance advantages of transparent huge pages. Huge pages, being performance critical in cloud applications with large memory footprints, especially in virtualized

cloud environments, need to be supported in this two-tier memory system. We present a new hot/cold classification mechanism to distinguish frequently accessed pages (hot) from infrequently accessed ones (cold). We implement Thermostat in Linux kernel version 4.5 and show that it can transparently move cold data to slow memory while satisfying a 3% tolerable slowdown. We show that our online cold page identification mechanism incurs no observable performance overhead and can migrate up to 50% of application footprint to slow memory while limiting performance degradation to 3%, thereby reducing memory cost up to 30%.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] "Top500 november 2016." https://www.top500.org/lists/2016/11/. [Online; accessed 10-Feb-2017].

[2] "Project catapult." https://www.microsoft.com/en-us/research/project/project-catapult/. [Online; accessed 10-Feb-2017].

[3] "Amazon ec2 instance types." https://aws.amazon.com/ec2/instance-types/. [Online; accessed 10-Feb-2017].

[4] "Apache spark." http://spark.apache.org/. [Online; accessed 10-Feb-2017].

[5] Intel and Micron, "A Revolutionary Breakthrough In Memory Technology." http://www.intel.com/newsroom/kits/nvm/3dxpoint/pdfs/Launch_Keynote.pdf, 2015. [Online; accessed 29-November-2015].

[6] L. A. Barroso, J. Clidaras, and U. Hoelzle, *The Datacenter as a Computer:An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool, 2013.

[7] NVIDIA Corporation, "Unified Memory in CUDA 6." http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/, 2013. [Online; accessed 28-May-2014].

[8] NVIDIA Corporation, "Compute Unified Device Architecture." https://developer.nvidia.com/cuda-zone, 2014. [Online; accessed 31-July-2014].

[9] AMD Corporation, "What is Heterogeneous System Architecture (HSA)?." http://developer.amd.com/resources/heterogeneous-computing/what-is-heterogeneous-system-architecture-hsa/, 2014. [Online; accessed 28-May-2014].

[10] NVIDIA Corporation, "NVIDIA Launches World's First High-Speed GPU Interconnect, Helping Pave the Way to Exascale Computing." http://nvidianews.nvidia.com/News/NVIDIA-Launches-World-s-First-High-Speed-GPU-Interconnect-Helping-Pave-the-Way-to-Exascale-Computin-ad6.aspx, 2014. [Online; accessed 28-May-2014].

[11] HyperTransport Consortium, "HyperTransport 3.1 Specification." http://www.hypertransport.org/docs/twgdocs/HTC20051222-0046-0035.pdf, 2010. [Online; accessed 7-July-2014].

[12] INTEL Corporation, "An Introduction to the Intel QuickPath Interconnect." http://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html, 2009. [Online; accessed 7-July-2014].

[13] N. Agarwal, D. Nellans, M. O'Connor, S. W. Keckler, and T. F. Wenisch, "Unlocking Bandwidth for GPUs in CC-NUMA Systems," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 354–365, February 2015.

[14] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler, "Page Placement Strategies for GPUs within Heterogeneous Memory Systems," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 607–618, March 2015.

[15] HSA Foundation, "HSA Platform System Architecture Specification - Provisional 1.0." http://www.slideshare.net/hsafoundation/hsa-platform-system-architecture-specification-provisional-verl-10-ratifed, 2014. [Online; accessed 28-May-2014].

[16] S. Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun, "A Case of System-level Hardware/Software Co-design and Co-verification of a Commodity Multi-processor System with Custom Hardware," in *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pp. 513–520, October 2012.

[17] D. Vantrease, M. H. Lipasti, and N. Binkert, "Atomic Coherence: Leveraging Nanophotonics to Build Race-free Cache Coherence Protocols," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 132–143, 2011.

[18] INTEL Corporation, "Intel Xeon Processor E5 v3 Family." http://ark.intel.com/products/family/78583/Intel-Xeon-Processor-E5-v3-Family#@All, 2015. [Online; accessed 16-April-2015].

[19] J. H. Kelm, M. R. Johnson, S. S. Lumettta, and S. J. Patel, "WAYPOINT: Scaling Coherence to Thousand-core Architectures," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 99–110, September 2010.

[20] D. Johnson, M. Johnson, J. Kelm, W. Tuohy, S. S. Lumetta, and S. Patel, "Rigel: A 1,024-Core Single-Chip Accelerator Architecture," *IEEE Micro*, vol. 31, pp. 30–41, July 2011.

[21] B. A. Hechtman, S. Che, D. R. Hower, Y. Tian, B. M. Beckmann, M. D. Hill, and D. A. Wood, "QuickRelease: A Throughput-oriented Approach to Release Consistency on GPUs," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 189–200, February 2014.

[22] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous System Coherence for Integrated CPU-GPU Systems," in *International Symposium on Microarchitecture (MICRO)*, pp. 457–467, December 2013.

[23] N. Agarwal, D. Nellans, E. Ebrahimi, T. F. Wenisch, J. Danskin, and S. W. Keckler, "Selective GPU caches to eliminate CPU-GPU HW cache coherence," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 494–506, March 2016.

[24] Intel, "Intel and Micron Produce Breakthrough Memory Technology." `https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/`, 2015. [Online; accessed 29-April-2016].

[25] N. Agarwal and T. F. Wenisch, "Thermostat: Application-transparent Page Management for Two-tiered Main Memory," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[26] "Department of Energy Awards $425 Million for Next Generation Supercomputing Technologies." `http://energy.gov/articles/department-energy-awards-425-million-next-generation-supercomputing-technologies`, 2014. [Online; accessed Sep-19-2016].

[27] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *International Symposium on Workload Characterization (IISWC)*, pp. 44–54, October 2009.

[28] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, v.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," tech. rep., IMPACT Technical Report, IMPACT-12-01, University of Illinois, at Urbana-Champaign, March 2012.

[29] J. Mohd-Yusof and N. Sakharnykh, "Optimizing CoMD: A Molecular Dynamics Proxy Application Study," in *GPU Technology Conference (GTC)*, March 2014.

[30] C. Chan, D. Unat, M. Lijewski, W. Zhang, J. Bell, and J. Shalf, "Software Design Space Exploration for Exascale Combustion Co-design," in *International Supercomputing Conference (ISC)*, pp. 196–212, June 2013.

[31] M. Heroux, D. Doerfler, J. Crozier, H. Edwards, A. Williams, M. Rajan, E. Keiter, H. Thornquist, and R. Numrich, "Improving Performance via Mini-applications," Tech. Rep. SAND2009-5574, Sandia National Laboratories, September 2009.

[32] J. Tramm, A. Siegel, T. Islam, and M. Schulz, "XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis," *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, September 2014.

[33] Hynix Semiconductor, "Hynix GDDR5 SGRAM Part H5GQ1H24AFR Revision 1.0." http://www.hynix.com/datasheet/pdf/graphics/H5GQ1H24AFR(Rev1.0).pdf, 2009. [Online; accessed 30-Jul-2014].

[34] JEDEC, "High Bandwidth Memory(HBM) DRAM - JESD235." http://www.jedec.org/standards-documents/docs/jesd235, 2013. [Online; accessed 28-May-2014].

[35] J. Y. Kim, "Wide IO2 (WIO2) Memory Overview." http://www.cs.utah.edu/events/thememoryforum/joon.PDF, 2014. [Online; accessed 30-Jul-2014].

[36] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated Memory for Expansion and Sharing in Blade Servers," in *International Symposium on Computer Architecture (ISCA)*, pp. 267–278, June 2009.

[37] INTEL Corporation, "Intel Xeon Processor E7-4870." http://ark.intel.com/products/75260/Intel-Xeon-Processor-E7-8893-v2-37_5M-Cache-3_40-GHz, 2014. [Online; accessed 28-May-2014].

[38] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch, "System-level Implications of Disaggregated Memory," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 1–12, 2012.

[39] M. M. K. Martin, M. D. Hill, and D. J. Sorin, "Why On-chip Cache Coherence is Here to Stay," *Communications of the ACM (CACM)*, vol. 55, pp. 78–89, July 2012.

[40] S. H. Pugsley, J. B. Spjut, D. W. Nellans, and R. Balasubramonian, "SWEL: Hardware Cache Coherence Protocols to Map Shared Data Onto Shared Caches," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 465–476, September 2010.

[41] J. F. Cantin, M. H. Lipasti, and J. E. Smith, "Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking," in *International Symposium on Computer Architecture (ISCA)*, pp. 246–257, June 2005.

[42] D. Sanchez and C. Kozyrakis, "SCD: A Scalable Coherence Directory with Flexible Sharer Set Encoding," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 1–12, February 2012.

[43] O. Villa, D. R. Johnson, M. O'Connor, E. Bolotin, D. Nellans, J. Luitjens, N. Sakharnykh, P. Wang, P. Micikevicius, A. Scudiero, S. W. Keckler, and W. J. Dally, "Scaling the Power Wall: A Path to Exascale," in *International Conference on High Performance Networking and Computing (Supercomputing)*, November 2014.

[44] J. Corbet, "Four-level page tables merged." http://lwn.net/Articles/117749/.

[45] AMD, "Secure virtual machine architecture reference manual," *Virtualization, AMD64*, 2005.

[46] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig, "Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization.," *Intel Technology Journal*, vol. 10, no. 3, 2006.

[47] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating Two-dimensional Page Walks for Virtualized Systems," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 26–35, 2008.

[48] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 163–174, April 2009.

[49] T. M. Aamodt, W. W. L. Fung, I. Singh, A. El-Shafiey, J. Kwa, T. Hetherington, A. Gubran, A. Boktor, T. Rogers, A. Bakhoda, and H. Jooybar, "GPGPU-Sim 3.x Manual." `http://gpgpu-sim.org/manual/index.php/GPGPU-Sim_3.x_Manual`, 2014. [Online; accessed 4-December-2014].

[50] J. Tuck, L. Ceze, and J. Torrellas, "Scalable Cache Miss Handling for High Memory-Level Parallelism," in *International Symposium on Microarchitecture (MICRO)*, pp. 409–422, December 2006.

[51] A. Minkin and O. Rubinstein, "Circuit and method for prefetching data for a texture cache." US Patent 6,629,188, issued September 20, 2003.

[52] Free Software Foundation, "GNU Binutils." `http://www.gnu.org/software/binutils/`, 2014. [Online; accessed 5-August-2014].

[53] J. D. McCalpin, "Memory Bandwidth and Machine Balance in Current High Performance Computers," *IEEE Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, December 1995.

[54] NVIDIA Corporation, "CUDA C Best Practices Guide." `https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#allocation`, 2014. [Online; accessed 28-July-2014].

[55] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal, "DiDi: Mitigating the Performance Impact of TLB Shootdowns Using a Shared TLB Directory," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 240–249, October 2011.

[56] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 743–758, March 2014.

[57] J. Power, M. Hill, and D. Wood, "Supporting x86-64 Address Translation for 100s of GPU Lanes," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 568–578, February 2014.

[58] J. D. McCalpin, "STREAM - Sustainable Memory Bandwidth in High Performance Computers." `http://www.cs.virginia.edu/stream/`. [Online; accessed 28-May-2014].

[59] AMD Corporation, "Compute Cores." `https://www.amd.com/Documents/Compute_Cores_Whitepaper.pdf`, 2014. [Online; accessed 15-April-2015].

[60] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth, "Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 381–394, March 2013.

[61] M. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. Loh, "Heterogeneous Memory Architectures: A HW/SW Approach For Mixing Die-stacked And Off-package Memories," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 126–136, February 2015.

[62] C. Chou, A. Jaleel, and M. K. Qureshi, "BATMAN: Maximizing Bandwidth Utilization of Hybrid Memory Systems," Tech. Rep. TR-CARET-2015-01, Georgia Institute of Technology, March 2015.

[63] M. Daga, A. M. Aji, and W.-C. Feng, "On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing," in *Symposium on Application Accelerators in High-Performance Computing (SAAHPC)*, pp. 141–149, July 2011.

[64] NVIDIA Corporation, "New NVIDIA TITAN X GPU Powers Virtual Experience "Thief in the Shadows" at GDC." `http://blogs.nvidia.com/blog/2015/03/04/smaug/`, 2015. [Online; accessed 16-April-2015].

[65] IBM Corporation, "POWER8 Coherent Accelerator Processor Interface (CAPI)." `http://www-304.ibm.com/webapp/set2/sas/f/capi/home.html`, 2015. [Online; accessed 16-April-2015].

[66] H. Chen and C. Wong, "Wiring And Crosstalk Avoidance In Multi-chip Module Design," in *Custom Integrated Circuits Conference*, pp. 28.6.1–28.6.4, May 1992.

[67] NVIDIA Corporation, "CUDA C Programming Guild v7.0." `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`, 2015. [Online; accessed 09-May-2015].

[68] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo Filter: Practically Better Than Bloom," in *International on Conference on Emerging Networking Experiments and Technologies*, pp. 75–88, December 2014.

[69] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An Improved Construction for Counting Bloom Filters," in *European Symposium on Algorithms (ESA)*, pp. 684–695, September 2006.

[70] A. R. Lebeck and D. A. Wood, "Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors," in *Proceedings of ISCA-22*, pp. 48–59, 1995.

[71] A.-C. Lai and B. Falsafi, "Selective, Accurate, and Timely Self-Invalidation Using Last-Touch Prediction," in *International Symposium on Computer Architecture (ISCA)*, pp. 139–148, June 2000.

[72] P. Stenstrom, "A Survey of Cache Coherence Schemes for Multiprocessors," *IEEE Computer*, vol. 23, pp. 12–24, June 1990.

[73] M. Huang, M. Mehalel, R. Arvapalli, and S. He, "An Energy Efficient 32-nm 20-MB Shared On-Die L3 Cache for Intel Xeon Processor E5 Family," *IEEE Journal of Solid-State Circuits (JSSC)*, vol. 48, pp. 1954–1962, August 2013.

[74] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services," in *International Symposium on Computer Architecture (ISCA)*, pp. 13–24, June 2014.

[75] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, R. Allmon, R. Rayess, S. Maresh, and J. Emer, "Triggered Instructions: A Control Paradigm for Spatially-programmed Architectures," in *International Symposium on Computer Architecture (ISCA)*, pp. 142–153, June 2013.

[76] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan, "Data Tiering in Heterogeneous Memory Systems," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, pp. 15:1–15:16, April 2016.

[77] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable High Performance Main Memory System Using Phase-change Memory Technology," in *International Symposium on Computer Architecture (ISCA)*, pp. 24–33, 2009.

[78] Y. Li, J. Choi, J. Sun, S. Ghose, H. Wang, J. Meza, J. Ren, and O. Mutlu, "Managing Hybrid Main Memories with a Page-Utility Driven Performance Model," *arXiv preprint arXiv:1507.03303*, 2015.

[79] J. Corbet, "AutoNUMA: the other approach to NUMA scheduling." `http://lwn.net/Articles/488709/`, 2012. [Online; accessed 29-May-2014].

[80] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of Symposium on Cloud Computing*, pp. 143–154, 2010.

[81] Lespinasse, Michel, "idle page tracking / working set estimation." https://lwn.net/Articles/460762/, 2011. [Online; accessed 29-April-2016].

[82] C. A. Waldspurger, "Memory Resource Management in VMware ESX Server," *SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 181–194, 2002.

[83] F. Guo, S. Kim, Y. Baskakov, and I. Banerjee, "Proactively Breaking Large Pages to Improve Memory Overcommitment Performance in VMware ESXi," in *International Conference on Virtual Execution Environments(VEE)*, pp. 39–51, 2015.

[84] Intel, "Intel 64 and IA-32 Architectures Developer's Manual," 2016. [Online; accessed 2-May-2016].

[85] AMD, "AMD-V Nested Paging ." http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf, 2008. [Online; accessed 2-May-2016].

[86] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient Virtual Memory for Big Memory Servers," in *International Symposium on Computer Architecture (ISCA)*, pp. 237–248, June 2013.

[87] M. Gorman, "Huge pages part 4: benchmarking with huge pages." http://lwn.net/Articles/378641/, 2010. [Online; accessed 11-Aug-2016].

[88] Menage, Paul, "CGROUPS ." http://lxr.free-electrons.com/source/Documentation/cgroup-v1/memory.txt. [Online; accessed 4-May-2016].

[89] J. Gandhi, A. Basu, M. H. Hill, and M. M. Swift, "A Tool to Instrument x86-64 TLB Misses," *SIGARCH Computer Architecture News (CAN)*, 2014.

[90] "Libvirt virtualization api." http://libvirt.org/formatdomain.html#elementsCPU. [Online; accessed 13-Aug-2016].

[91] Dickins, Hugh, "huge tmpfs: THPagecache implemented by teams." https://lwn.net/Articles/682623/, 2016. [Online; accessed 30-April-2016].

[92] "PerfKit Benchmarker." https://github.com/GoogleCloudPlatform/PerfKitBenchmarker. [Online; accessed 8-May-2016].

[93] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 37–48, 2012.

[94] "TPC-C." http://www.tpc.org/tpcc/. [Online; accessed 8-May-2016].

[95] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux, "Oltp-bench: An extensible testbed for benchmarking relational databases," *Proceedings of the VLDB Endowment*, vol. 7, no. 4, pp. 277–288, 2013.

[96] "Aerospike." `http://www.aerospike.com/`. [Online; accessed 8-Aug-2016].

[97] "Cassandra." `http://cassandra.apache.org/`. [Online; accessed 8-May-2016].

[98] "Redis." `http://redis.io/`. [Online; accessed 8-May-2016].

[99] "MemtableSSTable." `https://wiki.apache.org/cassandra/MemtableSSTable`. [Online; accessed 8-May-2016].

[100] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload Analysis of a Large-scale Key-value Store," in *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pp. 53–64, 2012.

[101] M. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing Lifetime and Security of PCM-Based Main Memory with Start-Gap Wear Leveling," in *International Symposium on Microarchitecture (MICRO)*, 2009.

[102] E. Kultursay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating STT-RAM as an Energy-efficient Main Memory Alternative," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 256–267, April 2013.

[103] S. Phadke and S. Narayanasamy, "MLP-Aware Heterogeneous Memory System," in *Design, Automation & Test in Europe (DATE)*, pp. 1–6, March 2011.

[104] J. Mogul, E. Argollo, M. Shah, and P. Faraboschi, "Operating System Support for NVM+DRAM Hybrid Main Memory," in *Workshop on Hot Topics in Operating Systems (HotOS)*, pp. 14–18, May 2009.

[105] R. A. Bheda, J. A. Poovey, J. G. Beu, and T. M. Conte, "Energy Efficient Phase Change Memory Based Main Memory for Future High Performance Systems," in *International Green Computing Conference (IGCC)*, pp. 1–8, July 2011.

[106] L. Ramos, E. Gorbatov, and R. Bianchini, "Page Placement in Hybrid Memory Systems," in *International Conference on Supercomputing (ICS)*, pp. 85–99, June 2011.

[107] K. Wilson and B. Aglietti, "Dynamic Page Placement to Improve Locality in CC-NUMA Multiprocessors for TPC-C," in *International Conference on High Performance Networking and Computing (Supercomputing)*, pp. 33–35, November 2001.

[108] W. Bolosky, R. Fitzgerald, and M. Scott, "Simple but Effective Techniques for NUMA Memory Management," in *Symposium on Operating Systems Principles (SOSP)*, pp. 19–31, December 1989.

[109] T. Brecht, "On the Importance of Parallel Application Placement in NUMA Multiprocessors," in *Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS)*, pp. 1–18, September 1993.

[110] R. LaRowe, Jr., C. Ellis, and M. Holliday, "Evaluation of NUMA Memory Management Through Modeling and Measurements," *IEEE Transactions on Parallel Distribibuted Systems*, vol. 3, pp. 686–701, November 1992.

[111] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum, "Operating System Support for Improving Data Locality on CC-NUMA Compute Servers," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 279–289, September 1996.

[112] R. Iyer, H. Wang, and L. Bhuyan, "Design and Analysis of Static Memory Management Policies for CC-NUMA Multiprocessors," *Journal of Systems Architecture*, vol. 48, pp. 59–80, September 2002.

[113] D. Tam, R. Azimi, and M. Stumm, "Thread Clustering: Sharing-aware Scheduling on SMP-CMP-SMT Multiprocessors," in *European Conference on Computer Systems (EuroSys)*, pp. 47–58, March 2007.

[114] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing Shared Resource Contention in Multicore Processors via Scheduling," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 129–142, March 2010.

[115] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn, "Using OS Observations to Improve Performance in Multicore Systems," *IEEE Micro*, vol. 28, pp. 54–66, May 2008.

[116] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova, "A Case for NUMA-aware Contention Management on Multicore Systems," in *USENIX Annual Technical Conference (USENIXATC)*, pp. 1–15, June 2011.

[117] M. Awasthi, D. Nellans, K. Sudan, R. Balasubramonian, and A. Davis, "Handling the Problems and Opportunities Posed by Multiple On-Chip Memory Controllers," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 319–330, September 2010.

[118] J. Zhao, G. Sun, G. Loh, and Y. Xie, "Optimizing GPU Energy Efficiency with 3D Die-stacking Graphics Memory and Reconfigurable Memory Interface," *ACM Transactions on Architecture and Code Optimization*, vol. 10, pp. 24:1–24:25, December 2013.

[119] B. Wang, B. Wu, D. Li, X. Shen, W. Yu, Y. Jiao, and J. Vetter, "Exploring Hybrid Memory for GPU Energy Efficiency Through Software-hardware Co-design," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 93–103, September 2013.

[120] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian, "CHOP: Integrating DRAM Caches for CMP Server Platforms," *IEEE Micro*, vol. 31, pp. 99–108, March 2011.

[121] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan, "Enabling Efficient and Scalable Hybrid Memories Using Fine-Granularity DRAM Cache Management," *IEEE Computer Architecture Letters*, vol. 11, pp. 61–64, July 2012.

[122] M. Swanson, L. Stoller, and J. Carter, "Increasing TLB Reach using Superpages Backed by Shadow Memory," in *International Symposium on Computer Architecture (ISCA)*, pp. 204–213, June 1998.

[123] B. Pham, A. Bhattacharjee, Y. Eckert, and G. Loh, "Increasing TLB Reach by Exploiting Clustering in Page Translations," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 558–567, February 2014.

[124] B. Gerofi, A.Shimada, A. Hori, T. Masamichi, and Y. Ishikawa, "CMCP: A Novel Page Replacement Policy for System Level Hierarchical Memory Management on Many-cores," in *International Symposium on High-performance Parallel and Distributed Computing (HPDC)*, pp. 73–84, June 2014.

[125] C. McCurdy, A. Cox, and J. Vetter, "Investigating the TLB Behavior of High-end Scientific Applications on Commodity Microprocessors," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 95–104, April 2008.

[126] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel, "Rigel: An Architecture and Scalable Programming Interface for a 1000-core Accelerator," in *International Symposium on Computer Architecture (ISCA)*, pp. 140–151, June 2009.

[127] M. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood, "Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors," *ACM Transactions on Computer Systems*, vol. 11, pp. 300–318, November 1993.

[128] L. Cheng, N. Muralimanohar, K. Ramani, R. Balasubramonian, and J. B. Carter, "Interconnect-Aware Coherence Protocols for Chip Multiprocessors," in *International Symposium on Computer Architecture (ISCA)*, pp. 339–351, June 2006.

[129] B. Falsafi, A. R. Lebeck, S. K. Reinhardt, I. Schoinas, M. D. Hill, J. R. Larus, A. Rogers, and D. A. Wood, "Application-Specific Protocols for User-Level Shared Memory," in *International Conference on High Performance Networking and Computing (Supercomputing)*, November 1994.

[130] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. Adve, V. Adve, N. Carter, and C.-T. Chou, "DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 155–166, 2011.

[131] H. Sung, R. Komuravelli, and S. V. Adve, "DeNovoND: Efficient Hardware Support for Disciplined Non-determinism," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 13–26, 2013.

[132] H. Sung and S. V. Adve, "DeNovoSync: Efficient Support for Arbitrary Synchronization Without Writer-Initiated Invalidations," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 545–559, 2015.

[133] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: Near-optimal Block Placement and Replication in Distributed Caches," in *International Symposium on Computer Architecture (ISCA)*, pp. 184–195, June 2009.

[134] B. A. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. F. Duato, "Increasing the Effectiveness of Directory Caches by Deactivating Coherence for Private Memory Blocks," in *International Symposium on Computer Architecture (ISCA)*, pp. 93–104, June 2011.

[135] A. Ros and S. Kaxiras, "Complexity-effective Multicore Coherence," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 241–252, September 2012.

[136] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W. Hwu, "An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 347–358, March 2010.

[137] S. Pei, M.-S. Kim, J.-L. Gaudiot, and N. Xiong, "Fusion Coherence: Scalable Cache Coherence for Heterogeneous Kilo-Core System," in *Advanced Computer Architecture*, vol. 451 of *Communications in Computer and Information Science*, pp. 1–15, Springer, 2014.

[138] S. Kaxiras and A. Ros, "A New Perspective for Efficient Virtual-cache Coherence," in *International Symposium on Computer Architecture (ISCA)*, pp. 535–546, June 2013.

[139] M. Alisafaee, "Spatiotemporal Coherence Tracking," in *International Symposium on Microarchitecture (MICRO)*, pp. 341–350, Decemeber 2012.

[140] A. Moshovos, "RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence," in *International Symposium on Computer Architecture (ISCA)*, pp. 234–245, June 2005.

[141] J. Zebchuk, E. Safi, and A. Moshovos, "A Framework for Coarse-Grain Optimizations in the On-Chip Memory Hierarchy," in *International Symposium on Microarchitecture (MICRO)*, pp. 314–327, Decemeber 2007.

[142] J. Huh, J. Chang, D. Burger, and G. S. Sohi, "Coherence Decoupling: Making Use of Incoherence," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 97–106, October 2004.

[143] D. R. Hower, B. A. Hechtman, B. M. Beckmann, B. R. Gaster, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous-race-free Memory Models," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 427–440, March 2014.

[144] B. R. Gaster, D. Hower, and L. Howes, "HRF-Relaxed: Adapting HRF to the Complexities of Industrial Heterogeneous Memory Models," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 1, pp. 7:1–7:26, 2015.

[145] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt, "Cache Coherence for GPU Architectures," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 578–590, February 2013.

[146] S. Kumar, A. Shriraman, and N. Vedula, "Fusion: Design Tradeoffs in Coherent Cache Hierarchies for Accelerators," in *International Symposium on Computer Architecture (ISCA)*, pp. 733–745, June 2015.

[147] T. Suh, D. Blough, and H.-H. Lee, "Supporting Cache Coherence In Heterogeneous Multiprocessor Systems," in *Design Automation Conference (DAC)*, pp. 1150–1155, February 2004.

[148] B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Communications of the ACM (CACM)*, vol. 13, pp. 422–426, July 1970.

[149] R. Pagh and F. F. Rodler, "Cuckoo Hashing," *Journal of Algorithms*, vol. 51, pp. 122–144, May 2004.

[150] K. Strauss, X. Shen, and J. Torrellas, "Flexible Snooping: Adaptive Forwarding and Filtering of Snoops in Embedded-Ring Multiprocessors," in *International Symposium on Computer Architecture (ISCA)*, pp. 327–338, June 2006.

[151] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos, "A Tagless Coherence Directory," in *International Symposium on Microarchitecture (MICRO)*, pp. 423–434, Decemeber 2009.

[152] H. Zhao, A. Shriraman, S. Dwarkadas, and V. Srinivasan, "SPATL: Honey, I Shrunk the Coherence Directory," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 33–44, October 2011.

[153] A. Moshovos, G. Memik, A. Choudhary, and B. Falsafi, "JETTY: Filtering Snoops for Reduced Energy Consumption in SMP Servers," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 85–96, January 2001.

[154] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the Conference on Programming Language Design and Implementation(PLDI)*, pp. 190–200, 2005.

[155] F. X. Lin and X. Liu, "Memif: Towards Programming Heterogeneous Memory Asynchronously," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 369–383, 2016.

[156] G. Chen, B. Wu, D. Li, and X. Shen, "PORPLE: An Extensible Optimizer for Portable Data Placement on GPU," in *International Symposium on Microarchitecture (MICRO)*, pp. 88–100, 2014.

[157] M. Ekman and P. Stenstrom, "A Cost-Effective Main Memory Organization for Future Servers," in *International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 45a–45a, April 2005.

[158] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar, "Dynamic Tracking of Page Miss Ratio Curve for Memory Management," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 177–188, 2004.

[159] Y. Baskakov, P. Gao, and J. K. Spencer, "Identification of Low-activity Large Memory Pages," May 2016. US Patent 9,330,015.

[160] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System Software for Persistent Memory," in *Proceedings of the European Conference on Computer Systems(EuroSys)*, pp. 15:1–15:15, 2014.

[161] S. Kannan, A. Gavrilovska, and K. Schwan, "pVM: Persistent Virtual Memory for Efficient Capacity Scaling and Object Storage," in *Proceedings of the European Conference on Computer Systems(EuroSys)*, pp. 13:1–13:16, 2016.

[162] R. Azimi, L. Soares, M. Stumm, T. Walsh, and A. D. Brown, "PATH: page access tracking to improve memory management," in *Proceedings of the International Symposium on Memory Management*, pp. 31–42, 2007.

[163] T. Walsh, "Generating Miss Rate Curves with Low Overhead Using Existing Hardware," Master's thesis, University of Toronto, 2009.

[164] S. Eranian, "What Can Performance Counters Do for Memory Subsystem Analysis?," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 26–30, 2008.