# DISAGGREGATED MEMORY MANAGEMENT

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Midhul Varma Vuppalapati

August 2026

DISAGGREGATED MEMORY MANAGEMENT

Midhul Varma Vuppalapati, Ph.D.

Cornell University 2026

The limitations of the traditional memory architecture in servers—where memory modules are tightly coupled to the processor via the DDR memory interconnect—have resulted in unsustainable memory capacity costs and the emergence of memory bandwidth bottlenecks in datacenters. Memory disaggregation is a promising architectural solution to the limitations of the traditional server-centric memory architecture—memory is decoupled from individual servers into a common pool accessible by multiple servers through alternate interconnect(s) different from DDR. Such an architecture enables statistically multiplexing memory capacity demands across multiple servers, thus improving utilization and minimizing costs. It also enables mitigating memory bandwidth limitations by leveraging alternate interconnects that offer better bandwidth scaling trends than the DDR memory interconnect.

The widespread deployment of cache-coherent interconnects such as Compute Express Link (CXL) has opened the gateway to realizing the vision of memory disaggregation. Such interconnects enable the processor to directly access disaggregated memory through load and store instructions in a cache-coherent fashion without the need for software-level interposition. While the hardware support exists, realizing memory disaggregation over such cache-coherent interconnects requires effective memory management mechanisms. This entails two core problems. First, since disaggregated memory typically has higher access latency than locally-attached memory, deployments only partially disag-

gregate memory to limit the performance impact on applications—each server retains a certain amount of local memory in addition to having access to the shared disaggregated memory. In such a setting, application performance critically depends on the placement of data across local and disaggregated memory. As application access patterns change over time, we need mechanisms that can dynamically adapt data placement to maximize application performance. Second, since workload memory capacity and bandwidth demands can vary drastically at fine-grained timescales, we need mechanisms that can dynamically (re)allocate disaggregated memory resources across workloads to achieve the improved utilization promise of memory disaggregation.

This dissertation makes four core contributions towards improved disaggregated memory management: (1) to understand the impact of memory disaggregation on application performance, we build an in-depth understanding of the interconnects within each server or host that facilitate memory accesses (that is, the host interconnects), (2) building atop our understanding of host interconnects, we characterize the impact of data placement across local and disaggregated memory on application performance, (3) we explore the design of OS memory management mechanisms that automatically adapt data placement to maximize application performance, (4) we explore mechanisms to (re)allocate disaggregated memory resources across tenants under dynamic demands.

Characterizing the impact of memory disaggregation on application performance requires first understanding the host interconnects that facilitate memory accesses. Memory accesses issued by processors and peripheral devices (*e.g.*, network interface cards, storage devices, GPUs) must first traverse the processor interconnect (*e.g.*, the on-chip Mesh in Intel processors) and/or the peripheral interconnect (*e.g.*, PCIe) before they reach the DDR memory interconnect

or a cache-coherent interconnect like CXL. We demonstrate that memory access performance is not limited merely by the characteristics of any individual interconnect (*e.g.*, bandwidth), but rather governed by the interplay between different interconnects. For example, we show that contention within the memory interconnect results in nanosecond-scale memory access latency inflation, which percolates through the processor and/or peripheral interconnects to impact the memory access throughput of processor cores and peripheral devices differently. We develop a conceptual abstraction to reason about the interplay between host interconnects and the resulting impact on application performance. Furthermore, we build an in-depth understanding of cache-coherent interconnects for memory disaggregation by studying the performance characteristics of CXL on modern hosts. We uncover the root causes of CXL memory access latency overheads and bandwidth inefficiencies through a combination of careful measurement and reverse-engineering of the underlying hardware.

Building atop our understanding of host interconnects, we characterize the impact of data placement on application performance. Discussions about memory disaggregation often focus on the performance degradation caused by offloading data to disaggregated memory. We show that memory disaggregation over cache-coherent interconnects like CXL can actually *improve* application performance. Under multiple in-flight memory requests, access latency of local memory can increase due to memory interconnect contention (even when memory bandwidth is far from saturated) and exceed the access latency of disaggregated memory over CXL-like interconnects. In this regime, offloading data to CXL-attached disaggregated memory improves application performance by mitigating overload of the DDR memory interconnect and leveraging the additional bandwidth offered by cache-coherent interconnects like CXL.

While improving application performance via memory disaggregation is feasible on modern hardware, existing operating system memory management mechanisms preclude such operating points. We demonstrate that these existing mechanisms achieve far from optimal application performance for memory disaggregation over cache-coherent interconnects like CXL. This is because existing mechanisms focus on packing the hottest application data in local memory. This is based on an implicit assumption that, despite serving the hottest pages, the access latency of local memory is always lower than that of disaggregated memory—an assumption that is easily violated in the realistic case of multiple in-flight requests. We introduce Colloid, a memory management mechanism that embodies the principle of balancing access latencies—data placement should be performed so as to balance the average (loaded) access latencies of local and disaggregated memory. To realize this principle, Colloid innovates on both memory access latency measurement mechanisms and data placement algorithms. We integrate Colloid with three state-of-the-art memory management systems and demonstrate its effectiveness across a variety of workloads.

Colloid maximizes application performance for a given allocation of disaggregated memory to each host. Realizing the utilization promise of memory disaggregation, however, requires dynamically (re)allocating disaggregated memory capacity across multiple hosts, applications or users. Our key observation is that the classical max-min fairness algorithm for resource allocation provides many desirable properties (*e.g.*, Pareto efficiency, strategy-proofness and fairness), but only under a strong assumption—that user demands are static over time. For the realistic case of dynamic user demands, we show that the max-min fairness algorithm loses one or more of these properties. We present Karma, a resource allocation mechanism for dynamic demands. The key technical contribu-

tion in Karma is a credit-based resource allocation algorithm: in each quantum, users donate their unused resources and are assigned credits when other users borrow these resources; Karma carefully orchestrates the exchange of credits across users (based on their instantaneous demands, donated resources and borrowed resources), and performs prioritized resource allocation based on users' credits. We theoretically establish Karma guarantees related to Pareto efficiency, strategy-proofness and fairness for dynamic user demands. We demonstrate that these properties translate well into practice through empirical evaluation.

In addition to advancing the status quo of disaggregated memory management, the contributions of this dissertation point to several avenues of future research at the intersection of operating systems, distributed systems and computer architecture, both in the context of memory disaggregation and more broadly. We conclude with a discussion of these avenues and our broader outlook on the design of future computer systems software and hardware.

## BIOGRAPHICAL SKETCH

Midhul Vuppalapati is a Ph.D. candidate in the Computer Science department at Cornell University advised by Prof. Rachit Agarwal. He holds a Master of Science in Computer Science from Cornell University, and a Bachelor of Technology in Computer Science and Engineering from the Indian Institute of Technology Guwahati. He is a recipient of the Cornell University Fellowship, an ACM SIGCOMM Best Student Paper award, and two Cornell Computer Science Outstanding Teaching Assistant awards.

To my mentors, friends and family.

## ACKNOWLEDGEMENTS

The journey behind this dissertation has had many ups and downs, but what I will cherish the most—and am most grateful for—is the opportunity to work with and learn from such an incredible set of people.

I owe a tremendous debt of gratitude to my advisor, Rachit Agarwal. Rachit has invested an extraordinary amount of time and energy in my growth—he has *always* made time for me—all the way from my first PhD project to the academic job search. He taught me the scientific approach to systems research and showed me the joy of following my curiosity—always encouraging me to dig deeper rather than settle for answers based on high-level intuition. His extensive feedback has helped greatly refine not just my technical skills but also my writing and communication. Rachit has always set high expectations and pushed me to reach my potential, yet has also been remarkably patient and empathetic through the many stressful periods I faced, including personal ones. He has been my biggest champion, opening doors and creating opportunities I could not have imagined at the outset. I hope to emulate, for my future students, the kind of advisor Rachit has been to me.

Cornell has provided an extremely supportive environment. I thank Andrew Myers and Zhiru Zhang for serving on my dissertation committee. Several faculty members made time to provide feedback on my job talk despite my last-minute requests, and I am grateful to all of them: Fred Schneider, Eva Tardos, Robbert van Renesse, Hakim Weatherspoon, Lorenzo Alvisi, Adrian Sampson and Karthik Sridharan. Becky Stewart deserves special thanks for handling the myriad of administrative issues over the years—she has truly been a life-saver! Corey Torres, too, has my thanks for entertaining my countless room reservation requests even when it was no longer her responsibility.

I have been extremely fortunate to work with several amazing faculty members both within and outside Cornell. Anurag Khandelwal, a close collaborator on many of my early PhD projects, taught me a great deal. Despite being a newly minted Assistant Professor at the time, he always made time to work through problems with me, sometimes spending hours helping me debug low-level issues. Asaf Cidon and Simon Peter were also collaborators on my early projects and helped shape my outlook on systems research; I am deeply grateful to both of them for their support during the academic job search. It has also been a privilege to collaborate with and learn from Eva Tardos, Christos Kozyrakis, Arvind Krishnamurthy, and Baris Kasikci.

Members of our research group—both present and former—have made this PhD immensely more enjoyable. I had an absolute blast working with Jaehyun Hwang during my first and second years. Jaehyun not only taught me how to hack on the Linux kernel, but also helped me feel less isolated during the pandemic through our high-bandwidth Slack discussions. Over the past two years, I have greatly missed the presence of Saksham Agarwal and Qizhe Cai—my partners in crime for five years. Our late-night sprints before deadlines, deep technical discussions, the many conferences we attended together and the countless meals we shared are memories I will cherish. Saksham's observations at Google sowed the seeds for Chapter 2 of this dissertation. Collaborating with Qizhe taught me a great deal about the Linux network stack, and I thank him for continuing to check up on me regularly. Throughout the academic job search, both of them offered support and guidance for which I am very grateful.

I've had the pleasure of working closely with Omar Eqbal ever since he joined Cornell. Omar continues to have a smile on his face no matter what—which I find both refreshing and inspiring. Watching him grow has been

extremely rewarding and reinforced my decision to pursue an academic career. I also enjoyed working and interacting with Shreyas Karbanda, Akhil Azhikodan and Benny Rubin. Beyond our group, I am thankful to two student collaborators—Kushal Babel for being a fantastic colleague and a close friend, and Giannis Fikioris, without whom the theoretical analysis in Chapter 5 of this dissertation would not have been possible.

On the industry side, I had a wonderful summer at Azure Research with Daniel Berger. Chapter 3 of this dissertation would not have been possible if not for Daniel's amazing ability to conjure any resource that I asked for. I greatly enjoyed our daily discussions and benefited from the opportunity to have industry impact. His support during the academic job search was invaluable, and I am deeply grateful for it. Thanks also to Karthik Kumar and Pantea Zardoshti for their valuable input, and to Liangchen Yu and Stefan Saroiu for generously providing access to their DRAM analyzer. I am grateful to Kim Keeton and Suli Yang for hosting me at Google, giving me the freedom to explore a new area, and for their patience and encouragement during the job search.

Graduate school would have been far lonelier without the friends I made along the way. Saravanan Kandasamy has been a constant companion for the longest time. I thank him for helping me navigate stressful situations, for checking on me and keeping in touch even when he was far away, and for all the food that he has brought me. Abhishek Vijay has been a great friend over the years, and I am grateful for all his support and companionship. I've also enjoyed the company of Abhishek Shetty, Tanuja Sawant, Lucas Kasser, and my fellow systems lab occupants: Florian Suri-Payer, Suraaj Kanniwadi, Ali Farahbakhsh, Shouxu Lin, and Alicia Yang. For their thoughtful feedback on my job talk, I thank Joshua Fan, Abhilash Dharmavarapu, Suraaj Kanniwadi, Ali

Farahbakhsh, Abhishek Vijay, Yixiao Du, Alicia Yang and Rishabh Madan. I also thank Jeff, Orion Smedley, Eliza Hong, Bennett Santora, Jonah Botvinnick-Greenhouse, Pascal Bodmer, and the others at Cornell's juggling club for all the fun times. Kenn and Madeline Young opened their beautiful home to me for seven years, for which I am deeply grateful.

Before Cornell, my trajectory was shaped by mentors at Microsoft Research India, where I was fortunate to be part of an incredible cohort of Research Fellows. I am grateful to Muthian Sivathanu for taking a chance on me despite my lack of prior research experience. Watching him at work inspired me to pursue a PhD. I also thank Kaushik Rajan and Bhargav Gulavani for their mentorship during that time, and Venkatesh Tamarapalli at IIT Guwahati for encouraging me to pursue research. My undergraduate friends Sai, Nikhil and Rahul have stayed in touch and celebrated my achievements, and I thank them for it. I am also thankful to the broader IIT-G gang for their companionship and for the many trips during my visits to India.

Finally, none of this would have been possible without my family. I thank my mother, Rama, for her endless sacrifices and her unconditional encouragement to pursue my dreams. My father, Srinivas, introduced me to electronics and computers at an early age, which helped set me on this path. My sister, Manojna, has been a constant source of love and support throughout; my cousins Anusha, Bindu and Sahaj and my grandparents have cheered me on every step of the way. I am endlessly grateful to all of you.

## TABLE OF CONTENTS

## LIST OF TABLES

## LIST OF FIGURES

CHAPTER 1

# INTRODUCTION

We begin by motivating memory disaggregation (§1.1). We then provide a broad overview of different approaches to memory disaggregation (§1.2) and discuss key technical challenges (§1.3). Next, we provide an overview of the key contributions of this dissertation and its organization (§1.4).

## 1.1 Memory disaggregation: Motivation

The traditional memory architecture in servers (Figure 1.1 (left)) consists of DRAM modules exposed to the processor via the DDR memory interconnect. Unfortunately, technology trends for this architecture have become stagnant, making it difficult to scale memory capacity and memory interconnect bandwidth cost-effectively—packaging constraints limit the number of pins available to the processor and electrical signaling constraints limit the data rate per pin and the number of DRAM modules per pin [160, 175, 196, 208, 225]. The former constrains memory bandwidth and the latter constrains memory capacity. These trends have resulted in two critical problems in datacenters: unsustain-

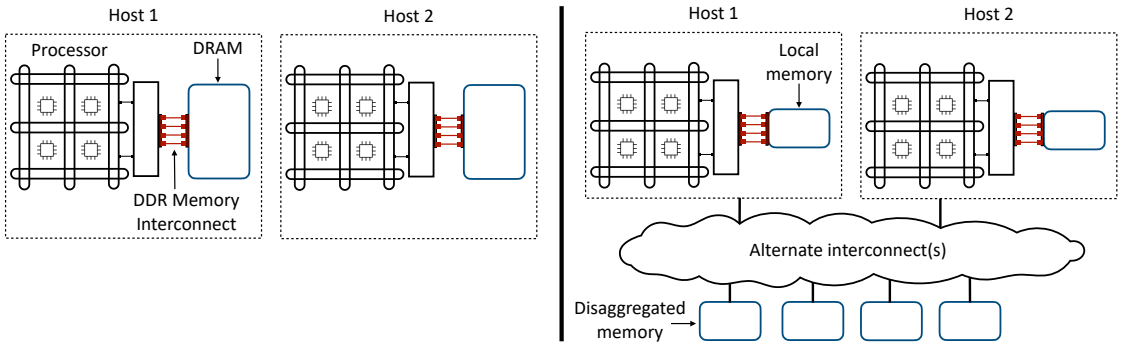



Figure 1.1: Traditional server-centric memory architecture (left) versus disaggregated memory architecture (right).

able memory capacity costs and the increasing prevalence of memory bandwidth bottlenecks.

**Unsustainable memory capacity costs.** Memory accounts for an increasingly large fraction of datacenter server costs (*e.g.*, ~37% for Meta [150] and ~50% for Microsoft Azure [157]). Much of this cost stems from suboptimal memory capacity utilization. First, workloads can have drastically different ratios of compute-to-memory capacity demands, while the traditional memory architecture tightly couples fixed ratios of cores to memory capacity within each server. This makes it difficult to pack workloads into servers to simultaneously maximize utilization of both compute and memory capacity. For instance, in Microsoft Azure, up to 36% of memory capacity is wasted when 90% of CPU cores are allocated [136]. Second, even without packing inefficiencies, workloads' memory capacity demands vary dynamically. For instance, memory capacity demands of Snowflake workloads vary by as much as 17× within minutes, and most workloads have demands whose standard deviation is 0.5 – 43× the average over time. Such dynamism results in poor average utilization because it forces provisioning memory capacity for peak demands—since the traditional memory architecture imposes a fixed per-server memory capacity limit, not provisioning sufficient memory capacity to accommodate peak demands results in having to either swap data to orders-of-magnitude slower storage or terminate workloads. For instance, Snowflake's system-wide average memory capacity utilization has been reported to be a mere 19% [218]. While statistically multiplexing demands across workloads can improve average utilization, the traditional memory architecture limits such multiplexing to workloads colocated on a single server.

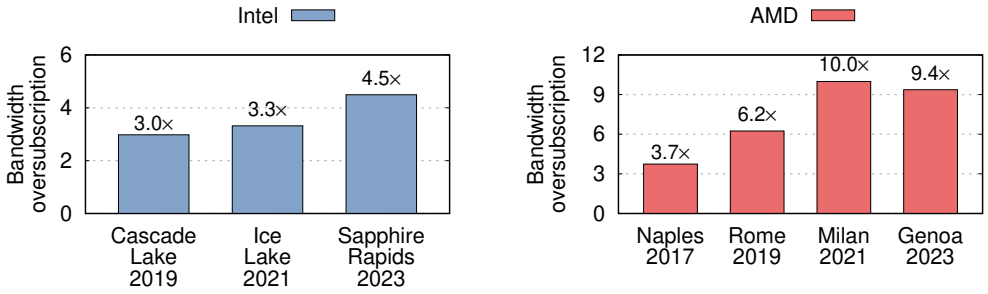



Figure 1.2: Memory bandwidth over-subscription trends for recent Intel (left) and AMD (right) processors. The y-axis shows maximum per-core load (assuming ideal memory access latency) multiplied by number of cores divided by memory bandwidth for each server generation.

**Emergence of memory bandwidth bottlenecks.** Memory interconnect bandwidth, already difficult to scale, is becoming increasingly oversubscribed in servers as processor core counts and per-core memory bandwidth usage continue to rise. Chiplet-based architectures drive the former. The latter stems from workloads becoming more data-intensive (*e.g.*, in Google's fleet, average per-core memory bandwidth usage of workloads increased by 1.4× from 2020 to 2024 [109]) and from each core generating more concurrent memory requests due to larger per-core buffers and more aggressive prefetchers [151, 214]. Figure 1.2 quantifies these oversubscription trends: recent Intel and AMD processors can generate as much as 4.5× and 9.37× more memory traffic than the available memory bandwidth. Indeed, several cloud providers have recently reported that memory bandwidth has emerged as a critical bottleneck that severely impacts application performance [3, 17, 109, 138, 142].

**Ideal vision of memory disaggregation.** Memory disaggregation is a promising architectural solution to the limitations of the traditional server-centric memory architecture. As illustrated in Figure 1.1 (right), memory is decoupled from individual servers into a common pool accessible by multiple servers through

alternate interconnect(s) different from DDR. First, such an architecture enables independently scaling compute and memory capacity to match workload demands, eschewing the packing constraints imposed by the traditional server-centric memory architecture. Second, it enables statistical multiplexing of demands across multiple servers, thus improving utilization under dynamic demands. Third, it enables mitigating the memory bandwidth limitations of the traditional server-centric memory architecture using alternate interconnects with better bandwidth scaling trends than DDR. Moreover, interconnect bandwidth can be scaled (with the number of servers) independently of memory capacity. Memory disaggregation also enables several additional benefits, such as improved flexibility in deploying memory by allowing memory modules of different generations and memory technologies to be deployed without having to replace hosts [57, 150, 246], decoupled fate sharing between compute and memory [37, 193], and low-overhead communication and synchronization across hosts [94].

## 1.2 Memory disaggregation: Approaches

Different approaches to realizing memory disaggregation can be characterized into two broad categories depending on the type of interconnect(s) used to expose disaggregated memory to the processor.

**Memory disaggregation over non-coherent interconnects.** Disaggregated memory is exposed over the datacenter network with the processor accessing it through a PCIe-attached Network Interface Card (NIC). Such disaggregation can be realized at different layers of the stack, each with different trade-offs.

OS-level mechanisms [7, 11, 86, 173, 182, 228, 235] treat disaggregated memory as a swap device and swap in and out data to and from local memory at page granularity underneath the virtual memory abstraction. This has the advantage of transparency, that is, it does not require modifications to applications, runtimes, and compilers. However, it introduces performance overheads because accessing data absent in local memory triggers page faults. Additionally, since data movement is restricted to page granularity, irregular accesses to objects smaller than the page size may fetch unnecessary data over the network. Application library and runtime level mechanisms [62, 190, 222, 223, 247] enable object granularity data movement and avoid the need for page faults. However, this comes at the cost of transparency. Despite having different trade-offs, both OS-level and application or runtime level mechanisms for disaggregation over PCIe-attached NICs share a common fundamental limitation: accessing data in disaggregated memory requires first loading it into local memory before the processor can access it with load and store instructions[1]. Therefore, while these mechanisms can improve memory capacity utilization, they cannot address memory bandwidth bottlenecks and can in fact exacerbate the problem because accessing objects in disaggregated memory generates additional local memory traffic.

**Memory disaggregation over cache-coherent interconnects.** An alternative approach to memory disaggregation leverages cache-coherent interconnects [13, 15, 50, 52, 54, 57, 97, 99, 172, 184, 203] to expose disaggregated memory directly to the processor. The prime example of such an interconnect is Compute Express Link (CXL), which is supported on all recent Intel and AMD processors and

[1]Optimizations like Intel DDIO [98] enable the NIC to write data to the processor's L3 cache. However, in the general case, one cannot guarantee that data is consumed by the processor before it is evicted to local memory [35]

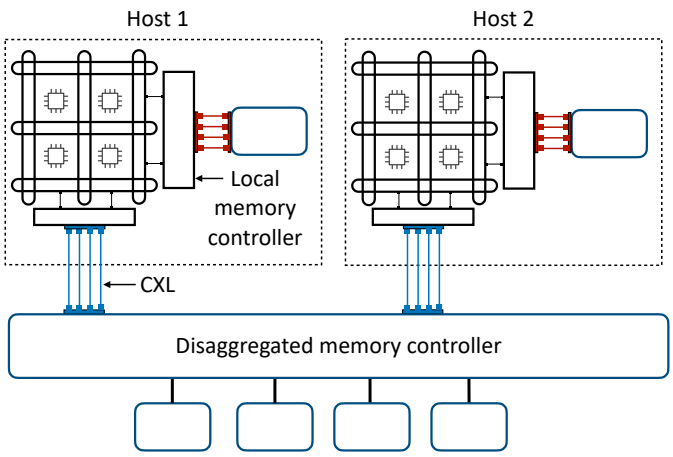



Figure 1.3: Illustration of memory disaggregation over CXL. Hosts are connected through CXL links to a disaggregated memory controller which is in turn connected to disaggregated memory modules.

has been deployed at scale in Azure and Meta. Figure 1.3 illustrates CXL-based memory disaggregation. This approach overcomes the fundamental limitation of disaggregation over non-coherent interconnects, where data must first be fetched into local memory before the processor can access it. Instead, disaggregated memory can be mapped into the processor's physical address space and accessed directly with load and store instructions in a cache-coherent fashion. This has two implications. First, accessing disaggregated memory no longer requires software-level interposition, reducing the overhead of memory disaggregation (*e.g.*, OS-level mechanisms do not need to incur page faults on the critical path). Second, disaggregated memory accesses are no longer limited by the bandwidth of the local DDR interconnect, allowing us to address memory bandwidth bottlenecks by tapping the additional bandwidth provided by the cache-coherent interconnect. CXL, for instance, builds on the serial interconnect technology of PCIe, enabling significantly higher bandwidth per pin than DDR, a parallel interconnect (PCIe Gen 5 achieves ~4× the bandwidth per pin of DDR5, and the gap continues to widen [45]).

## 1.3 Memory disaggregation: Technical challenges

Cache-coherent interconnects like CXL enable moving closer to the ideal vision of memory disaggregation. However, several technical challenges need to be resolved to make this a reality.

**Memory management between local and disaggregated memory.** While cache-coherent interconnects obviate the need for software-level interposition (*e.g.*, page faults) in the critical path, disaggregated memory still has higher access latency relative to local memory. For example, on state-of-the-art servers, CXL-exposed memory has 2× higher minimum access latency than local DDR-attached memory [57, 208, 246]. Larger scale disaggregation setups may require additional components in the memory access datapath which additionally increase latency (*e.g.*, retimers that increase the length of CXL links and switches that interconnect multiple disaggregated memory controllers). As a result, the performance of applications running atop disaggregated memory architectures depends critically on the placement of data across local and disaggregated memory [150, 246]. Moreover, the access patterns of applications can vary over time. To that end, we need efficient memory management mechanisms that dynamically adapt data placement to maximize application performance. This can be decomposed into multiple sub-challenges. First, we need efficient and accurate mechanisms to track applications' access patterns. Second, we need efficient mechanisms to move data between local and disaggregated memory. Third, we need algorithms that decide what subset of data to place in each kind of memory given access pattern information.

**Sharing disaggregated memory under extremely dynamic demands.** Real

workloads exhibit time-varying memory resource demands. For example, in Snowflake, memory capacity demands of jobs can vary by as much as 2× within tens of seconds [218]. Memory bandwidth demands can vary at sub-microsecond timescales [4]. Achieving the improved utilization promise of memory disaggregation requires (re)allocating disaggregated memory capacity and bandwidth across tenants at fine-grained timescales. To that end, we need resource allocation mechanisms for dynamic demands that maximize overall utilization while providing isolation across tenants.

**Designing disaggregated memory hardware and interconnect topologies.** As illustrated in Figure 1.3, memory disaggregation over cache-coherent interconnects requires a disaggregated memory controller that interconnects hosts to memory modules. Designing high-performance and low cost disaggregated memory controllers is important to ensure that the benefits of memory disaggregation are not outweighed by the costs. At the very least, a disaggregated memory controller needs to perform (1) protocol conversion (since hosts speak CXL and memory modules speak DDR) (2) mapping requests from input host ports to output memory module ports and (3) scheduling of requests to memory modules. This is challenging because the controller needs to process requests at massive line rates while operating under tight area and power constraints necessary to minimize cost. For example, an Intel Granite Rapids host can issue as many as 4 billion memory accesses per second (using 64 PCIe Gen5 lanes [105]), giving the disaggregated memory controller 250 picoseconds of processing time per-request to maintain line rate operation. Furthermore, scaling disaggregated memory to a larger number of hosts may require designing interconnect topologies to connect multiple disaggregated memory controllers to hosts.

**Cross host cache-coherence, consistency and synchronization.** Beyond multiplexing memory capacity for improved utilization, one can also use disaggregated memory to share data between hosts. Doing so requires synchronization mechanisms to maintain consistency. One approach is to enforce hardware-level cache-coherence across all hosts connected to a pool of disaggregated memory. For example, the CXL 3.0 specification allows implementing such coherence [57]—in Figure 1.3, hosts can issue requests for exclusive ownership of cachelines to the disaggregated memory controller, which in turn can send "back-invalidations" to other hosts which have cached copies of the data. Hardware-level cache-coherence across all hosts allows software to use existing multi-thread synchronization primitives (*e.g.*, mutexes, semaphores) for cross-host synchronization. However, the feasibility of such hardware-level cross-host cache-coherence remains unclear (*e.g.*, real hardware is yet to implement CXL 3.0 back invalidation support, if ever). In the absence of hardware cache-coherence across hosts, one is left with multiple coherence domains each with a subset of one or more hosts. Maintaining consistency in this regime requires designing software-level synchronization mechanisms and primitives that do not assume global cache-coherence.

**Fault tolerance.** In the traditional server-centric memory architecture, each host is a unit of failure. Disaggregated memory architectures lead to new failure modes. Consider the example in Figure 1.3. A host can fail without its data stored in disaggregated memory being necessarily lost. A disaggregated memory module can fail independently of the hosts. The disaggregated memory controller and any of the interconnect links could also fail. A naive failure model would entail treating all the hosts and disaggregated memory as a single unit of failure. While simple, this may be unnecessarily inefficient and/or expensive.

For example, failure of even a single disaggregated memory module would render all hosts unavailable. When a host fails, one misses the opportunity to resume work on a different host using state available in disaggregated memory. Overall, this motivates the need to revisit classical server-centric failure models and design fault-tolerance mechanisms to handle failures in disaggregated memory architectures.

**Security.** Memory disaggregation introduces additional security challenges. First, we need mechanisms to ensure that mutually distrustful tenants cannot access each other's data in disaggregated memory unless explicitly shared (akin to the protection provided by traditional virtual memory). Second, shared hardware resources such as the disaggregated memory controller in Figure 1.3 are likely to introduce avenues for new side channel attacks. Understanding and designing mechanisms to mitigate such side channel attacks is therefore important.

## 1.4 Thesis contributions and organization

This thesis makes the following contributions towards improved disaggregated memory management:

- **Understanding host interconnects and their impact on application performance:** To understand the impact of data placement across local memory and disaggregated memory on application performance, we build an in-depth understanding of the interconnects within the host that facilitate memory accesses. This includes the processor interconnect (*e.g.*, on-chip

Mesh network in Intel processors), peripheral interconnect (*e.g.*, PCIe), the traditional DDR memory interconnect (for local memory accesses) and alternate cache-coherent interconnects (for disaggregated memory accesses). In particular:

- We show that contention within the memory interconnect impacts application performance very differently depending on the source of memory requests (processor core or peripheral device) and their type (read or write). Deeper analysis reveals that this is because memory access throughput (of processor cores and peripheral devices) is not limited merely by memory interconnect bandwidth, but rather governed by the interplay between the memory interconnect and the processor and peripheral interconnects. We develop a conceptual abstraction to understand such interplay allowing us to explain how the effects of memory interconnect contention percolate via the processor and/or peripheral interconnects to impact application performance. Each host interconnect implements credit-based flow control mechanisms to ensure lossless data transfers. Memory interconnect contention results in nanosecond-scale latency inflation (even when memory bandwidth is far from saturated). Intuitively, such latency inflation can be tolerated without incurring throughput degradation if the processor and/or peripheral interconnect links have sufficient credits (which control the number of in-flight requests); however, when credits are exhausted, latency inflation results in interconnect bandwidth underutilization and throughput degradation. We present this work in Chapter 2.
- We build an in-depth understanding of cache-coherent interconnects

for memory disaggregation by studying the performance characteristics of CXL memory accesses on a recent Intel Granite Rapid processor with commercially available CXL memory controllers. First, we uncover the root causes of CXL memory access latency overheads through a combination of measurement and reverse-engineering. We find that CXL's higher latency relative to DDR is not merely due to it being a serial interconnect, but also due to poor interplay between CXL and existing processor interconnect locality abstractions. Second, we clarify misconceptions related to CXL tail latencies. We find that P99 and P99.9 tail memory access latencies are not an artifact of CXL hardware, but rather due to DRAM refreshes that periodically block memory accesses. Third, we show that CXL can achieve peak bandwidth comparable to that of DDR while using a significantly smaller number of processor pins. For example, in our setup, we find that CXL achieves 2.91× and 4.49× higher bandwidth per processor pin for read-only and read-write traffic respectively. Read-write traffic, in particular, benefits from CXL's ability to simultaneously utilize both directions of the PCIe link, while over DDR it is limited by being able to transmit in only one direction at any point of time. We present this work in the first part of Chapter 3.

- **Characterizing impact of data placement on application performance:** Building upon our understanding of host interconnects, we characterize the impact of data placement across local and disaggregated memory on application performance. We demonstrate that offloading data to CXL-exposed disaggregated memory can actually *improve* application performance by up to 1.8×. This is because, under multiple in-flight re-

quests, DDR access latency can increase by up to 4.8× the DDR latency under a single in-flight request, due to interconnect contention, causing it to exceed the minimum CXL access latency by up to 2.14×. In such a regime, offloading data to CXL reduces the load on the DDR memory interconnect—thereby reducing its access latency—and improving application performance by leveraging the additional bandwidth offered by CXL. We present this work in the second part of Chapter 3.

- **OS memory management over cache-coherent interconnects:** We show that existing OS memory management mechanisms achieve far from optimal application performance for memory disaggregation over cache-coherent interconnects like CXL. This is because existing mechanisms focus on packing the hottest application data in local memory. This is based on a deeply entrenched assumption that, despite serving the hottest pages, the access latency of local memory is always lower than that of disaggregated memory—an assumption that is easily violated in the realistic case of multiple in-flight memory requests even under moderate loads. In the regime where local memory access latency exceeds that of disaggregated memory, we demonstrate that performance of state-of-the-art memory management systems can be as much as 2.3× worse than the optimal. We introduce Colloid, a memory management mechanism that embodies the principle of balancing access latencies—data placement should be performed so as to balance the average (loaded) access latencies of local and disaggregated memory. To realize this principle, Colloid innovates on both memory access latency measurement mechanisms, and data placement algorithms. For access latency measurement, Colloid builds on our in-depth understanding of host interconnects—processors fundamentally

require a hardware component that routes memory requests to either local or disaggregated memory based on their physical address; furthermore, requests must remain queued at this component until an end-to-end response is received to maintain state that determines which processor core to send the response to; to that end, Colloid leverages queueing information at the routing component to gain low-overhead end-to-end visibility into access latency with nanosecond-scale precision. We integrate Colloid with three state-of-the-art memory management systems that cover a wide range of the design space (*e.g.*, different access tracking and page migration mechanisms), and demonstrate its effectiveness across a variety of workloads. We present this work in Chapter 4.

- **Sharing disaggregated memory resources under dynamic demands:** We consider the problem of allocating disaggregated memory resources across multiple hosts or applications or users. Through analysis of production workloads, we show that dynamic resource demands are the norm (*e.g.*, across Snowflake and Google workloads, we find that memory capacity demands vary by as much as 17× within minutes, with a majority of workloads having demands with standard deviation 0.5 – 43× of the average over time). Under dynamic demands, we show that the widely-deployed classical max-min fairness algorithm for resource allocation loses one or more of its key desirable properties: Pareto efficiency, strategy-proofness, and fairness. Building on this observation, we present Karma, a resource allocation mechanism for dynamic demands. The key contribution of Karma is a credit-based resource allocation algorithm: users receive credits when they donate a part of their fair share of resources (*e.g.*, if their demand is less than their fair share); users can

use these credits to borrow resources in the future when their demand is higher than their fair share. Karma carefully orchestrates resources and credits between donors and borrowers: donors are prioritized so as to keep credits across users as balanced as possible, and borrowers are prioritized so as to keep the resource allocation as fair as possible. We theoretically establish Karma guarantees for dynamic demands: (1) we show Karma guarantees Pareto efficiency at all times (2) For strategy-proofness, we show that Karma guarantees that a selfish user cannot increase their aggregate resource allocation by over-reporting their demands. In addition, we show a new surprising phenomenon (that may primarily be of theoretical interest): if a user had perfect knowledge about the future demands of all other users, the user can increase its own aggregate allocation by a small constant factor by under-reporting its demand; however, any imprecision in this future knowledge could lead to significant loss in the user's aggregate allocation (3) for fairness, we prove that given a set of (past) allocations, Karma guarantees an optimally-fair resource allocation. We demonstrate that these theoretical properties translate well into practice by evaluating an implementation of Karma on top of an existing disaggregated memory system on production workload traces. We present this work in Chapter 5.

The above contributions include material from several previously published collaborative works [214,215,217,218]. During my PhD I have published several other collaborative works that are not part of this dissertation [35,36,96,216].

CHAPTER 2

# UNDERSTANDING THE MEMORY INTERCONNECT AND ITS INTERPLAY WITH PROCESSOR AND PERIPHERAL INTERCONNECTS

In this chapter, we characterize how the memory interconnect, and its interplay with the processor and/or peripheral interconnects, impacts application performance (§2.2). We then provide the necessary background needed to build a deeper understanding of the interplay between these interconnects (§2.3). Next, we introduce a conceptual abstraction to study the interplay between host interconnects (§2.4), apply this lens to explain our observations from §2.2 (§2.5), and quantitatively validate our analysis (§2.6). Finally, we discuss related work (§2.7) and the implications of this chapter to the rest of the thesis (§2.8).

## 2.1 Overview

The processor, memory, and peripheral interconnects work together to enable data transfer between devices (processors, memory, network interface cards, storage devices, accelerators, etc.) within a host. In this chapter, we refer to the collection of these interconnects as *the host network*. Several studies from large-scale production datacenters [3, 138, 142] have demonstrated that contention within the host network can result in significant throughput degradation, tail latency inflation, and isolation violation for networked applications. As eloquently argued in [3, 4, 35, 138], the host network is becoming an increasingly prominent bottleneck due to unfavorable technology trends: performance of peripheral interconnects is improving much more rapidly than processor and memory interconnects, resulting in increasing imbalance of resources and contention within the host network. Designing future network protocols, operat-

ing systems, and hardware requires an in-depth understanding of the various regimes and root causes of such contention within the host network.

The key idea that drives our study is domain-by-domain credit-based flow control, a conceptual abstraction to study the host network. We demonstrate that the host network can be decomposed into multiple "domains" (sub-network of the host network)[1], each of which uses an independent credit-based flow control mechanism. Specifically, the sender of each domain is assigned credits that are used to limit the amount of data the sender can inject into the domain; the sender consumes a credit to send one message, and this credit is replenished when the message receipt is acknowledged by the receiver of the domain. Different domains within the host network have different numbers of credits and different unloaded latency. Each data transfer, depending on the source (compute or peripheral device) and on the type (read or write), traverses a different set of domains. The end-to-end performance for each transfer depends on the number of credits and the per-request latency of domains traversed by that transfer. Many details of existing host network hardware are not public; nevertheless, we reverse engineer Intel host architecture to characterize each domain, its credits, and its unloaded latency.

The lens of domain-by-domain credit-based flow control enables us to capture the subtle interplay between processor, memory, and peripheral interconnects that leads to nanosecond-scale latency inflation and host resource underutilization in certain domains. This, along with the knowledge of domains traversed by each data transfer, allows us to near-precisely explain how nanosecond-scale inefficiencies within the host network percolate through the

[1]The notion of a domain here is different from administrative domains in the Internet architecture, and in ATM networks [10, 49, 71, 74]. Importantly, unlike administrative domains, the domains in our study do not need to be non-overlapping.

host hardware, operating systems, and network protocols to negatively impact application-level performance. We do this both for applications generating peripheral-to-memory traffic (referred to as P2M apps, *e.g.*, networked and storage applications [38, 59, 204, 239, 243]) and for applications generating compute-to-memory traffic (referred to as C2M apps, *e.g.*, in-memory databases [65, 70, 186, 202] and systems for graph analytics [81, 128, 148]). More concretely:

- We reproduce the phenomenon of contention within the host network and its impact on networked applications observed in previous studies. We do this for both networked applications using in-kernel [4] and hardware-offloaded RoCE/PFC [138, 142] transport protocols. We find that networked applications (P2M apps in our case) can indeed suffer from performance degradation, *e.g.*, when both C2M apps and networked applications are doing memory writes. We provide precise root causes for the phenomenon. We also extend observations made in all prior studies [3, 4, 138, 142]: we demonstrate (and provide explanation for) degradation in C2M app performance along with networked app performance.
- We identify new, previously unreported, regimes of contention within the host network: we find that—in sharp contrast to the phenomenon observed in previous studies [3, 4, 138, 142]—P2M apps can, in fact, cause severe performance degradation for C2M apps for most workloads, with minimal or no impact on the P2M app performance. For instance, we observe that when C2M apps are colocated with P2M apps performing storage operations, the C2M app suffers from 1.2 – 2× performance degradation, with no impact on the P2M app performance. These new regimes are reproducible across multiple generations of servers with different processors, different memory band-

width to core count ratios, and different configurations (*e.g.*, with and without direct cache access [98], with and without prefetching, etc.).

Our study suggests that the impact of contention within the host network has implications that are more far-reaching than the context of networked applications considered in previous studies [3, 4, 138, 142]. In particular, all our observations hold even when all applications are contained within a single host.

In the context of the remainder of this dissertation, contention within the host network and our contributions towards better understanding the host network have several implications for disaggregated memory management, which we will discuss in §2.8 and further elaborate on in Chapters 3 and 4. More broadly, our work may be of independent interest to researchers and practitioners not only in computer networking but also in operating systems and computer architecture. The code, along with the documentation necessary to reproduce our results, is available at `https://github.com/host-architecture/understanding-the-host-network`.

## 2.2 Impact on application performance

In this section, we broadly characterize the interplay of processor, memory, and peripheral interconnects within the host network using four "quadrants" (§2.2.1) that reveal different regimes in terms of contention within the host network and performance degradation for C2M and P2M apps. Our key findings are:

- The first regime, referred to as the blue regime, captures a new phenomenon:

C2M apps observe performance degradation, while P2M apps observe minimal or no performance degradation. Surprisingly, this phenomenon can happen even when memory bandwidth is far from saturated.

- The other regime, referred to as the red regime, captures the phenomenon observed in previous studies [3, 4, 138, 142]: P2M apps observe severe performance degradation when memory bandwidth gets saturated. In addition, we find that C2M apps also observe significant performance degradation.

This section focuses on characterizing the host contention regimes; we discuss the root causes in §2.5. We first focus on a setup where all traffic is contained within the host (P2M traffic generated by locally attached storage devices)—this allows us to isolate the impact of contention within the host network from the impact of network protocol behavior (packet drops, queueing delays, and/or PFC pause frames) on application performance. We then discuss how our observations generalize to networked applications.

We use two testbeds with different processors and different resource ratios (Table 2.1). The first testbed uses Intel Ice Lake processors and has roughly the same resource ratios (cores, memory bandwidth and PCIe bandwidth) as the testbed in the Google study [3]. The second testbed uses Intel Cascade Lake processors and has a lower core to memory bandwidth ratio. We run our experiments on a single socket. Each DRAM module is attached through a separate channel, and a simple sequential read microbenchmark saturates more than 90% of theoretical maximum memory bandwidth.

We first present the new phenomenon: C2M apps observing performance degradation, with minimal or no performance degradation for P2M apps. This phenomenon is reproducible for a variety of configurations: multiple C2M apps

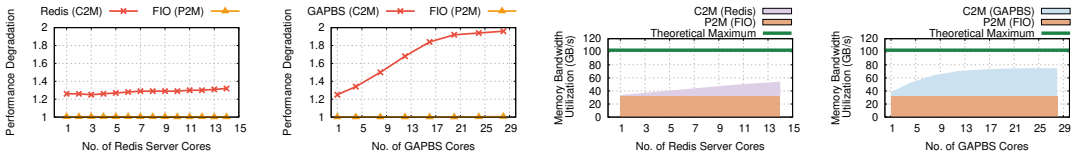



Figure 2.1: A new phenomenon of contention within the host network: C2M and P2M apps are colocated, C2M app performance degrades while P2M app performance is unaffected. This happens even though cores are isolated and memory bandwidth is far from saturated. (a-d; left-right) (a, b) Performance degradation observed by C2M and P2M when they are colocated (ratio of isolated throughput and colocated throughput for each data point; degradation for GAPBS is the slowdown—ratio of colocated execution time to isolated execution time); (c, d) Memory bandwidth utilization when C2M and P2M are colocated, broken down by C2M and P2M.

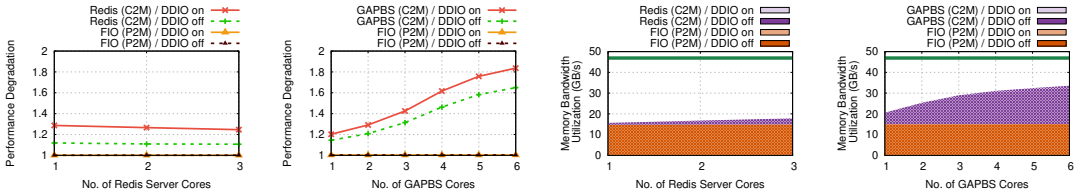



Figure 2.2: Enabling DDIO can worsen performance degradation for both C2M and P2M applications when the working set size does not fit in cache (left-right, a-d): (a, b) Performance degradation for C2M and P2M when they are colocated with DDIO on/off; (c, d) Memory bandwidth utilization when C2M and P2M are colocated with DDIO on/off.

with different compute-to-memory bandwidth demands, multiple server configurations, with and without Intel Data Direct I/O (DDIO) [98] technology, and with and without prefetching.

| | **Ice Lake** | **Cascade Lake** |
|---|---|---|
| **CPU** | Xeon Platinum 8362 | Xeon Gold 6234 |
| **Cores** | 32 @ 2.8GHz | 8 @ 3.3GHz |
| **LLC** | 48MB | 24MB |
| **DRAM** | 4×3200MHz DDR4 | 2×2933MHz DDR4 |
| **DRAM BW** | 102.4GB/s | 46.9GB/s |
| **PCIe** | 8×PM173X NVMe | 4×P5800X NVMe |
| **PCIe BW** | 32GB/s | 16GB/s |

Table 2.1: Hardware configuration of our two servers. All the specifications are for a single socket in each server. DRAM and PCIe bandwidths are theoretical maximum values.

**C2M and P2M apps used in our experiments.** We use two C2M apps, each with different compute-to-memory bandwidth demands and with different access patterns. The first C2M app is a popular in-memory database called Redis [186], and the second C2M app is a standard graph processing framework called the GAP Benchmark Suite (GAPBS) [27]. GAPBS is more memory bandwidth intensive and performs lighter-weight computations than Redis.

For Redis, we use the standard sharding-based multi-core deployment setup [39]—multiple independent Redis server instances (each with its own keyspace) running on a dedicated set of cores. Clients run on a different set of dedicated cores (1 client core per server core) and issue queries to the server instances using Unix domain sockets (the most efficient inter-process communication mechanism supported by Redis [187]). We use the standard YCSB-C (100% read) workload with a uniform random access pattern; as is standard, clients issue queries with parallelism given by the knee-point of the latency-throughput curve. Performance is measured in terms of the throughput (queries/sec). The working set size per server core is 1 million key-value pairs with a 1KB value size, exceeding the system Last-Level Cache (LLC) even for a single server core. As a result, the observed cache miss ratio is >95%, and a large number of C2M memory reads are generated. For GAPBS, we run the PageRank workload on a random graph of $2^{25}$ nodes and degree 16, using the GAPBS default parameters. Performance is measured in terms of execution time (lower is better). A single graph instance is shared across all the cores. The workload has a ~5GB memory footprint, significantly larger than the cache, resulting in a large number of random C2M memory reads. We focus on non-cache-resident memory-intensive workloads since the phenomenon was observed in datacenters for similar workloads [3, 138].

We use a lightweight storage-based P2M app, FIO [22], that performs storage accesses with minimal computational overhead. We configure it to perform sequential reads with 8MB request sizes. This is representative of storage workloads that perform large sequential operations, for example, storage nodes of distributed data stores [204] for analytics workloads. The performance metric is throughput measured in IOPS. The reads result in direct memory access (DMA) writes to host memory from the storage device, leading to a large volume of P2M write traffic. While DDIO minimizes P2M traffic for many workloads by servicing DMA requests from the cache instead of memory, it is well known that it is not effective for all workloads [35,67,210]. Our P2M workload is in the latter category—due to the large sequential requests, the application buffers do not fit into the small portion of the cache that DDIO is allowed to use [67], thus leading to cache misses and evictions/writebacks for every DMA in steady state. Therefore, we observe nearly the same average memory bandwidth utilization for this workload with/without DDIO.

In the following experiments, we first run each of the C2M and P2M apps in isolation. We then colocate them and measure the performance degradation for each app. We start with the Ice Lake setup (Table 2.1). We partition cores between the applications by pinning each to a separate set of cores—we dedicate 4 cores to the P2M app (which is more than sufficient to saturate PCIe bandwidth without compute being a bottleneck) and run the C2M app on the remaining cores with a varying number of cores. We enable DDIO and hardware prefetching on this setup.

**C2M app performance degrades even when memory bandwidth is far from saturated.** When Redis (C2M) and FIO (P2M) are colocated, as shown in Fig-

ure 2.1(a), Redis observes throughput degradation (1.25 – 1.32×) while FIO remains unaffected. The surprising observation here is that degradation is observed even though cores and PCIe bandwidth are isolated across the applications and memory bandwidth utilization is far from saturation as shown in Figure 2.1(c) (ranging between 33 – 53% of the theoretical maximum, and the utilization curve has not flattened out). Although the LLC is shared between the applications, it does not play much of a role in determining performance, since both C2M and P2M traffic observe nearly 100% cache miss ratio even when they are run in isolation. We investigate the root cause of this performance degradation in §2.5.

Using GAPBS rather than Redis results in the same high-level observation—C2M performance degrades (1.28 – 1.98×) while P2M remains unaffected (Figure 2.1(b)). This again happens even when memory bandwidth is far from saturated (as shown in Figure 2.1(d) for fewer than 15 GAPBS cores). As one would intuitively expect, the magnitude of performance degradation is larger for GAPBS compared to Redis since it is more memory intensive—Redis spends only a part of its time stalled on memory accesses, while GAPBS is stalled on memory accesses nearly all of the time.

**Impact of prefetching, processor generations, and resource ratios.** Given the random access nature of the Redis and GAPBS workloads, hardware prefetchers have little to no impact on performance. For both workloads, we found $< 5\%$ difference in performance when comparing prefetch on/off configurations in both isolated and colocated cases. We repeat the above experiments on the Cascade Lake setup (Table 2.1). Here, we dedicate 2 cores to FIO and run the C2M app on the remaining cores. The corresponding results are shown in the

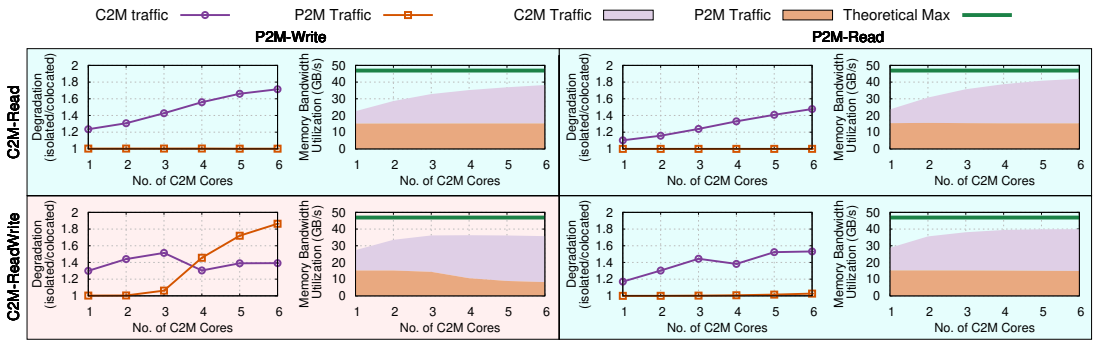



Figure 2.3: Blue and red regimes across four quadrants (1-4 are shown on top-left, top-right, bottom-left, bottom-right respectively). Quadrants are shaded with the color of the regime they show. For each quadrant, the left column shows the throughput degradation observed by C2M and P2M (ratio of throughput when run in isolation to the throughput when colocated) and the right column shows the memory bandwidth utilization when they are colocated broken down by C2M and P2M traffic.

(DDIO on) curves of Figure 2.2. We see the same general observation as in the Ice Lake setup—C2M app performance degrades while the P2M app observes no degradation, thus showing that our observations apply across different processor generations and resource ratios.

**DDIO can worsen performance degradation when app working set size does not fit in cache.** As discussed previously, DDIO is not effective for our P2M workload and does not reduce its memory bandwidth utilization when run in isolation. To study if DDIO has any second-order impact when C2M/P2M apps are colocated, we re-ran the above experiments with DDIO disabled on our Cascade Lake setup (this was not possible on our Ice Lake setup because DDIO is permanently enabled there). The corresponding results, shown in Figure 2.2, reveal a surprising observation: DDIO results in worse performance degradation for C2M apps for both Redis and GAPBS (Figures 2.2(a), 2.2(b)). This is surprising because our C2M workloads already have ~100% cache miss ratio even when run in isolation, and thus should ideally not be impacted by cache evictions caused by DDIO. We do not know how to explain this observation.

### 2.2.1 Blue and red contention regimes

Our previous experiments use real open-sourced apps that have fixed memory access patterns. We now switch to using lightweight apps with easy to control memory access patterns, enabling us to perform a deeper characterization of performance degradation trends and to study different combinations of read-/write for C2M/P2M apps.

**Workloads.** To generate C2M traffic, we use a modified version of the STREAM [152] benchmark that supports different read/write ratios. We use two C2M workloads: (1) a read-only workload that sequentially reads data from a 1GB buffer (using 64-byte AVX512 load instructions). This results in 100% memory reads (*C2M-Read*). (2) a write workload that sequentially writes data to a 1GB buffer (using 64-byte AVX512 store instructions). This generates 50% read and 50% write memory traffic since every cacheline is first read into the CPU's cache before the store instruction can be serviced and is later written back to memory during cache eviction (*C2M-ReadWrite*). For P2M, we run FIO with (1) 100% storage reads which translates to 100% memory writes (*P2M-Write*) and (2) 100% storage writes which translates to 100% memory reads (*P2M-Read*). Both our C2M and P2M workloads perform sequential accesses.

We run experiments on our Cascade Lake setup while colocating each of the two C2M workloads with each of the two P2M workloads. This results in a total of four scenarios, which we refer to as the four "quadrants". We disable prefetching and DDIO for better explicability. We found that while enabling each of these leads to different absolute degradation numbers, the trends and takeaways remain the same (when memory bandwidth is not saturated,

prefetching improves C2M throughput in both the isolated and colocated cases, but their ratio remains roughly the same). The degradation observed in the quadrants is shown in Figure 2.3. We classify the observations into two key regimes:

**Blue regime: C2M throughput degrades while P2M throughput does not.** In quadrant 1 (C2M-Read, P2M-Write), we observe 1.2 – 1.7× degradation in C2M throughput while P2M throughput remains unaffected. This degradation happens when memory bandwidth is far from saturated (for example, with a single C2M core), and increasing load leads to worse C2M throughput. Similarly, in quadrant 2 (C2M-Read, P2M-Read), while C2M throughput degrades, P2M throughput remains unaffected. C2M throughput degradation is lower than quadrant 1. Quadrant 4 (C2M-ReadWrite, P2M-Read) observes the same trend as in quadrant 2.

**Red regime: Both C2M and P2M throughput degrade.** Quadrant 3 (C2M-ReadWrite, P2M-Write) shows a range of different performance degradation trends. With 2 or fewer C2M cores, similar to quadrants 1 and 2, C2M throughput degrades while P2M throughput does not. For 3 C2M cores and above, once memory bandwidth is saturated, we see a completely different trend—C2M traffic now antagonizes P2M by getting an increasingly larger share of the memory bandwidth with increasing load, leading to larger throughput degradation for P2M traffic compared to C2M traffic. This captures the observations reported by recent works [3, 4, 138]. P2M traffic, however, does not get starved at higher load—with 5 and 6 C2M cores, we observe a relative stabilization of the memory bandwidth shares of C2M and P2M traffic.

The characterization of host network contention into blue and red regimes gen-

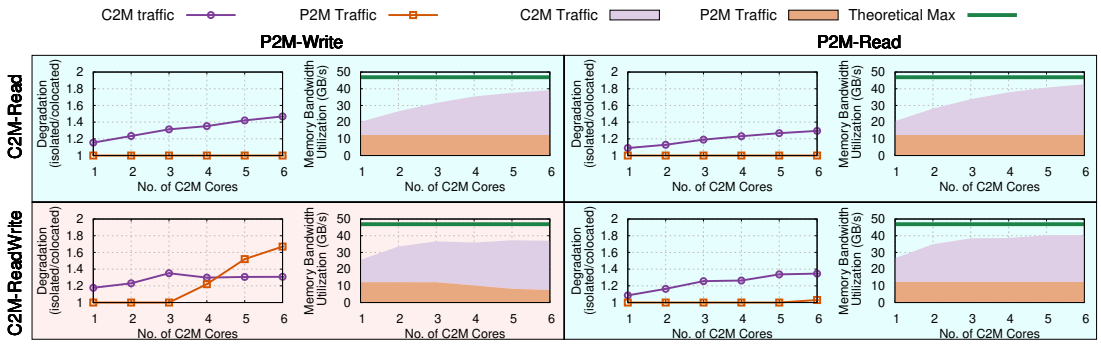



Figure 2.4: Blue and red regimes across four quadrants in the RDMA case study (1-4; clockwise). Each quadrant is shaded with the color of the regime it shows. For each quadrant, (left) shows throughput degradation observed by C2M and P2M workloads (ratio of throughput when run in isolation to the throughput when colocated) and (right) shows memory bandwidth utilization when they are colocated broken down by C2M and P2M traffic.

eralizes to cases where P2M traffic is generated by a NIC instead of local storage devices, and networked transfers use either kernel-based (Linux DCTCP) or hardware-offloaded transport (RDMA) mechanisms. We now present these observations.

**RDMA.** Our setup consists of two servers with RDMA-enabled NICs (Nvidia/Mellanox ConnectX-5) connected directly to each other through a 100Gbps network link. Each of the servers has the same Cascade Lake setup. We use a standard RDMA configuration of RoCE v2 with PFC enabled and MTU size of 4096. RDMA traffic is generated using the standard `ib_write_bw` and `ib_read_bw` tools from the RDMA perftest benchmarking suite. The former generates P2M write traffic and the latter generates P2M read traffic on the server-side. On the server-side, we colocate C2MRead and C2MReadWrite workloads. We disable CPU prefetchers and DDIO for better explainability.

With the above setup, we conduct experiments for all C2M/P2M read-/write combinations analogous to the four quadrants in Figure 2.3. Figure 2.4

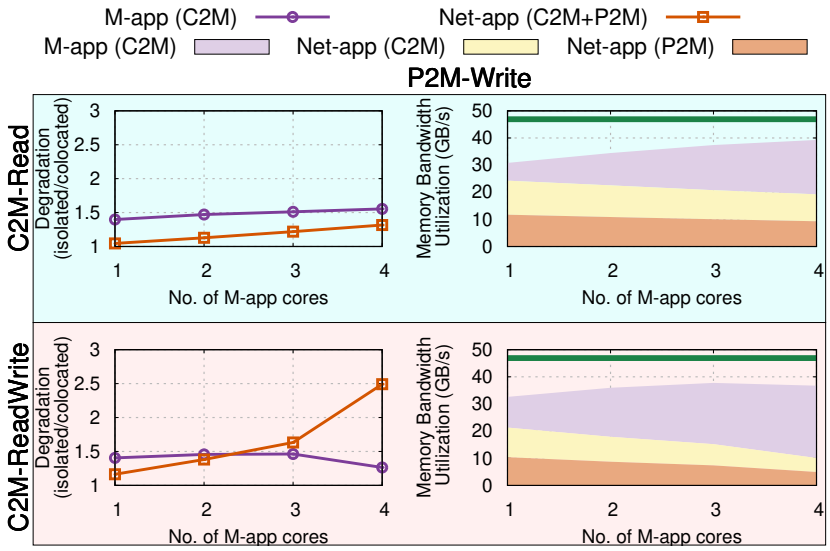



Figure 2.5: TCP case study results (a-d; clockwise). (a, b) respectively show throughput degradation observed when C2MRead and TCP Rx are colocated, and corresponding breakdown of memory bandwidth utilization (c, d) respectively show throughput degradation observed when C2MReadWrite and TCP Rx are colocated, and corresponding breakdown of memory bandwidth utilization.

shows the corresponding results. We observe the same performance degradation trends as in Figure 2.3. In particular, Quadrants 1, 2, 4 exhibit the blue regime (where C2M degrades but P2M does not), and Quadrant 3 exhibits the red regime (where both C2M and P2M throughput degrade). The magnitudes of C2M throughput degradation (in Quadrants 1-4) and P2M throughput degradation (in Quadrant 3) are slightly lower than what is observed in §2.2.1. This is because the NIC generates a slightly lower P2M load than the SSDs (with no memory contention, the SSDs generate ~112Gbps, while the NIC generates ~98Gbps).

**DCTCP.** To characterize host network contention regimes with Linux DataCenter TCP (DCTCP) over lossy fabric, we use a similar setup as in [4]: Two servers each with the Cascade Lake setup are connected via a 100Gbps link. Both run Linux kernel version 5.4.0 with DCTCP and all optimizations enabled (TSO/-

GRO, aRFS, Jumbo frames with 9K MTU). We send traffic from one server to another by running the standard `iperf` tool on 4 CPU cores on the sender and receiver each (this is sufficient to saturate 100Gbps link when there is no memory contention). Each sender core sends one long flow to its corresponding receiver core. Similar to [4], we focus on the receiver-side (Rx) as that is where the key bottlenecks are [35]. We colocate the C2MRead and C2MReadWrite workloads on the receiver-side server where P2M write traffic is generated by the NIC. CPU prefetchers and DDIO are kept disabled for better explainability.

Figure 2.5 shows the corresponding results. We again observe the same blue and red regimes, although the observed application-level performance trends are slightly different. In particular, the networked application observes performance degradation in both regimes. This is because, in addition to P2M traffic, the networked application also generates C2M traffic due to the data copy between application buffers and kernel socket buffers. In the blue regime, C2M throughput degradation slows down data copy processing, resulting in CPU bottleneck; this results in DCTCP flow control kicking in, reducing the P2M traffic load. In the red regime, P2M throughput degrades, but no congestion signal is sent back to the sender until packets are dropped at the NIC; this results in further throughput degradation, latency inflation, and violation of isolation properties as outlined in [3, 4].

Given that both the setups—P2M traffic from within the host, and P2M traffic from datacenter network transfers—lead to similar observations, we focus on the former setup for the rest of the chapter. It makes it easier to describe all our results.

## 2.3 Background on host interconnects

In this section, we provide a brief primer on the host network architecture and the datapath for C2M and P2M requests.

Figure 2.6 shows the host architecture: it consists of cores (with private L1/L2 caches), the LLC, the Caching and Home Agent (CHA), the Integrated IO controller (IIO), and the Memory Controller (MC) all connected by the on-chip processor interconnect. The CHA abstracts away the LLC and memory from the rest of the system while maintaining cache coherence[2]. Peripheral devices are attached to the IIO through the peripheral interconnect (typically PCIe). DRAM consists of a set of modules (Dual Inline Memory Modules, or DIMMs), each of which is attached to the MC through a memory channel. These memory channels constitute the memory interconnect. For simplicity, in Figure 2.6, we show a single module attached through a single memory channel.

**C2M datapath.** CPU-to-memory reads are generated by cores upon a cache miss through the following data path:

1. Upon an L1 cache miss, an entry is allocated in the core's Line Fill Buffer (LFB), and a request is sent to the L2 cache.
2. Upon reaching the L2 cache, the cacheline is either served from the L2 cache upon a cache hit (and the LFB entry is freed) or the request is sent to the CHA upon a cache miss.
3. The CHA serves the cacheline from the LLC if there is an LLC hit (and the

[2]Both the CHA and LLC are physically distributed into multiple slices. Based on the physical address, memory requests are routed to the correct slice. For simplicity, we represent the CHA/LLC as a single logical entity.

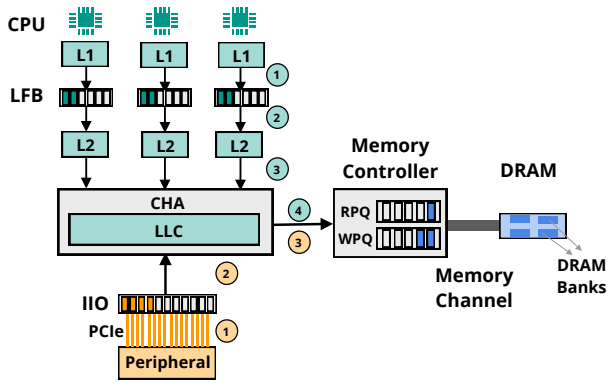



Figure 2.6: Host network architecture, and C2M and P2M datapaths. Details in §2.3.

LFB entry is freed). Otherwise, the CHA sends the request to the MC, where it is queued in the Read Pending Queue (RPQ).

4 The MC fetches the required cacheline from DRAM over the memory channel and returns the data back to the core while populating the caches and ultimately freeing the corresponding LFB entry.

Memory writes are generated upon cache evictions and follow a similar datapath: a write-back from the L2 cache is sent to the CHA, which either services it from the LLC or sends a write request to the MC, where it is queued in the Write Pending Queue (WPQ). The MC eventually issues the write to DRAM over the memory channel. *Importantly, unlike reads, writes generated by cores are asynchronous*: the CPU only has to wait for the request to be admitted to the CHA, and the CHA only has to wait for the request to be admitted to the WPQ.

**P2M datapath.** Each peripheral-to-memory request (read/write) incurs the following datapath:

1 Peripheral device initiates a DMA request to the IIO, that allocates an entry in the IIO (read/write) buffer per cacheline.

(2) IIO forwards the requests (at cacheline granularity) to the CHA. If DDIO is enabled and there is a cache hit, the CHA serves the request from the LLC (and the IIO buffer entry is freed). Otherwise, the CHA sends the request to the MC, where it is enqueued in the RPQ/WPQ.

(3) The MC serves read/write requests in a manner similar to the C2M datapath. After a read is serviced from DRAM, the data is returned to the IIO, at which point the IIO buffer entry is cleared and the data is sent back to the peripheral device. For writes, the IIO only needs to wait until the request is admitted to the WPQ before freeing its buffer entry.

The interconnects within the host physically implement hop-by-hop flow control mechanisms to ensure losslessness [100]. In the peripheral interconnect, this is implemented through the exchange of PCIe credits between the peripheral device and the IIO [4, 168]. The peripheral device needs a PCIe credit to send a request to the IIO; this credit is replenished once the corresponding IIO buffer entry is freed. In the memory interconnect, flow control happens implicitly through DRAM timing constraints [121, 130]. In the processor interconnect, implementation details of credit exchange are not public.

**DRAM operation.** MC reads/writes cachelines from/to DRAM over memory channels. Each memory channel can only transmit data in one direction (either reads or writes) at any point in time. The MC, therefore, operates in two separate modes, read mode and write mode, and maintains separate queues for reads and writes (RPQ and WPQ, respectively) per memory channel. Due to electrical constraints, switching between modes takes a certain delay (called the switching delay) during which the channel is idle [130]. The data in each DRAM module is organized into multiple banks. Each bank has multiple rows, each of

which stores a fixed number of cachelines, and a row buffer that can buffer a single row at any time. In order to access a cacheline, its corresponding row needs to be present in the corresponding bank's row buffer. If not, this results in a row miss, which incurs additional processing delay at the banks: The row needs to be loaded into the row buffer using an Activate (ACT) operation. If the row buffer contains a different row (i.e., row conflict), then it needs to be flushed using a Precharge (PRE) operation before a new row can be loaded, which incurs additional overhead.

For the remainder of the chapter, we focus on the scenario where C2M/P2M requests result in misses at all levels of the cache hierarchy (as is the case in the §2.2 experiments).

## 2.4 An abstraction to study the interplay between host interconnects

This section presents domain-by-domain credit-based flow control, a conceptual abstraction that captures the interplay between the processor, memory, and peripheral interconnects within the host network. We start by describing the abstraction along with the various domains and their characteristics in §2.4.1. We then describe in §2.4.2 how we reverse engineered the Intel host architecture to characterize each domain, its credits, and its unloaded latency.

### 2.4.1 Domain-by-domain credit-based flow control

We begin by defining domain-by-domain credit-based flow control. The host network is logically decomposed into multiple domains, each of which is a sub-network of the host network. Each domain uses an independent credit-based flow control mechanism. Specifically, the sender of each domain is assigned credits that are used to limit the number of in-flight requests that the sender can inject into the domain; the sender consumes a credit to send one request, and this credit is replenished when the request is acknowledged by the receiver of the domain. Depending on the number of credits, at any given point of time, there can be multiple concurrent in-flight requests within each domain.

Intuitively, domain-by-domain credit-based flow control generalizes the two flow control mechanisms studied in classical computer networking literature. On the one hand, end-to-end flow control mechanisms (*e.g.*, used in TCP and in receiver-driven datacenter transport protocols [34, 75, 89, 93]) are a special case where the entire path between a sender-receiver pair is a single domain. On the other hand, hop-by-hop credit-based flow control mechanisms (*e.g.*, used in ATM networks [110, 126, 127] and PFC-enabled RDMA networks [139, 248]) are a special case where each hop along the path between the sender-receiver pair is a domain.

Different domains within the host network have different numbers of credits and different unloaded latencies. Each request, depending on the source (compute or peripheral device) and on the type (read or write), traverses a different set of domains. Figure 2.7 shows the domains within the host network for each of the C2M and P2M read/write datapaths, with cores, peripheral devices, and DRAM as endpoints and intermediate components (i.e., LFB, IIO, CHA, and

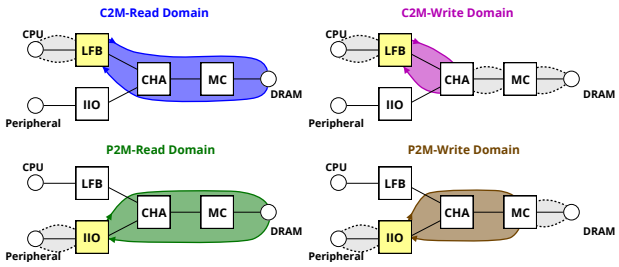



Figure 2.7: Domain-by-domain credit-based flow control in the host network. Different shaded regions within each sub-figure depict independent domains within the host network for respective C2M and P2M read/write datapaths. Data is transmitted between the CPU/Peripheral and the DRAM by traversing each domain using an independent credit-based flow control mechanism. The specific domain highlighted in color for each datapath is particularly interesting: these are the domains that will turn out to be the bottleneck within individual datapaths. Different domains can span different numbers of hops, leading to different domain latencies. Further, the number of domain credits (limited by the node marked as yellow) is also different for P2M vs C2M domains.

MC) as network nodes. We now discuss the four domains that turn out to be the most important ones (in that, these will be the "bottleneck" domains in individual datapaths):

- **C2M-Read Domain:** This domain spans all hops from LFB to DRAM. For each request, credit is allocated at the LFB and replenished once the request is serviced by DRAM and the data is returned to the LFB.
- **P2M-Read Domain:** This domain spans all hops from IIO to DRAM. For each request, credit is allocated at the IIO and replenished once the request is serviced by DRAM and the data is returned to the IIO.
- **C2M-Write Domain:** This domain spans only a single hop from LFB to CHA. For each request, credit is allocated at the LFB and replenished once the request reaches the CHA.
- **P2M-Write Domain:** This domain spans two hops from IIO to MC. For each request, credit is allocated at the IIO and replenished once the request reaches the MC.

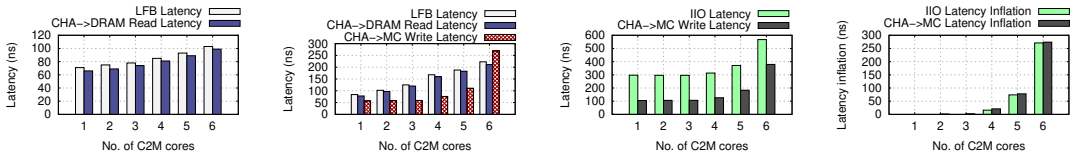



Figure 2.8: Evidence for domains, and per-domain characteristics (a-d, left-right). All y-axis values are on average. Discussion in §2.4.2.

The maximum throughput ($T$) for any domain is bound by $T \leq \frac{C \times 64}{L}$, where $C$ is a constant representing the hardware-specific number of credits available to the sender in the domain (in terms of cachelines), 64 is cacheline size in bytes, and $L$ is a variable representing the latency required to traverse all hops within the domain. Each of these factors can be different depending on the domain:

**Domain Credits (C).** The number of credits for the C2M-Read and C2M-Write domains is limited by the LFB size. The number of credits for the P2M-Write domain is limited by the IIO write buffer size. The number of credits for the P2M-Read domain is limited by the IIO read buffer size. For our servers, these numbers are 10 – 12, ~92, and >164 cachelines, respectively.

**Domain Latency (L).** Different domains span a different subset of network hops; this could result in different domains having different latencies for two reasons. First, simply due to spanning a different subset of network hops, different domains may have different unloaded latencies. Second, when C2M and P2M traffic contend for host network resources, queueing at the contention point may have different impact on different domains (only those domains are impacted that contain the contention point). As a result, contention within the host network may result in latency inflation for some domains but not others.

Given the number of credits and latency for a domain, the maximum through-

put of that domain is given by the expression $T \leq \frac{C \times 64}{L}$, as discussed above. The overall end-to-end throughput of a particular C2M or P2M app is the minimum throughput across all domains along the datapath for that app.

### 2.4.2 Evidence on domains and their characteristics

We now present evidence for domains and their characteristics, including a discussion of how we reverse-engineered several of the details by piecing together information from processor manuals [100,102] and conducting careful measurements.

**Measurement Methodology.** We use the Intel uncore performance monitoring counters [100] to capture average queue/buffer occupancy ($O$) and average request arrival rate ($R$) metrics at different nodes in the host network. In particular, we program the counters so that their values are aggregated in hardware every clock cycle and sample them at runtime in software every 1 second, which entails very low overhead. To compute average latency, we apply Little's law on the measured $O$ and $R$ values ($L = O/R$). We use the umask and opcode filtering capabilities of CHA counters to classify requests based on their source (CPU/Peripheral) and type (read/write), allowing us to capture all the above metrics on a per-domain basis.

**C2M-Read.** The C2M-Read domain spans all hops from LFB to DRAM because an LFB entry (and corresponding credit), once allocated, is only freed (and corresponding credit replenished) once the memory read request is serviced from DRAM and returned to the core to prevent duplicate memory requests to the

same cacheline [102, 211]. To validate this, we perform latency measurements while running the C2M-Read workload (§2.2.1) with varying number of cores. Figure 2.8(a) shows the measured LFB latency (time between allocation and replenishment of an LFB credit) alongside the CHA→DRAM read latency (time taken for request to traverse from CHA to DRAM and for response to return to CHA). As is evident from the figure, the LFB latency is always strictly greater than the CHA→DRAM read latency. Further, the inflation in LFB latency from 1 to 6 cores near perfectly matches inflation in CHA→DRAM read latency. This shows that the LFB latency is inclusive of the CHA→DRAM read latency, thus providing evidence that the C2M-Read domain includes all hops until DRAM. In all of our experiments, the maximum measured LFB occupancy is between 10 – 12 providing evidence that this is the number of domain credits (also corroborated in [61]). The unloaded domain latency is ~70ns, as is evident from the single-core data point in Figure 2.8(a).

**C2M-Write.** It is clear that the C2M-Write domain includes the LFB, the CHA (since each domain must span at least two nodes), and that it does not include DRAM (since writes are serviced to DRAM asynchronously [100]). The key challenge lies in determining whether the MC is part of the domain. To do so, we perform latency measurements while running the C2M-ReadWrite workload (§2.2.1) with varying number of cores. For this workload, the LFB latency is equal to the sum of the C2M-Read and C2M-Write domain latencies (and thus must be strictly greater than each of them). If the C2M-Write domain included the MC, then the C2M-Write latency (and consequently LFB latency) would always be strictly greater than the CHA→MC write latency (time taken for the request to traverse from CHA to MC). However, as shown in Figure 2.8(b), the CHA→MC write latency can exceed the LFB latency (e.g., with 6 C2M cores),

thus implying that the C2M-Write domain does not include the MC. Subtracting the unloaded C2M-Read domain latency from the LFB latency at the single core data point in Figure 2.8(b) gives us an estimate of ~10ns unloaded latency for the C2M-Write domain.

**P2M-Write.** To understand P2M-Write domain, we run a low-load P2M workload performing 4KB storage read requests with queue depth of 1, colocated with C2M-ReadWrite workload. Figure 2.8(c) shows the IIO latency (time between credit allocation and replenishment at IIO) alongside the CHA→MC write latency. We make three observations. First, the unloaded domain latency is ~300ns. Second, the IIO latency is always larger than the CHA→MC write latency. Finally, the inflation in IIO latency with increasing load near perfectly matches the inflation in CHA→MC write latency (Figure 2.8(d)), indicating that IIO latency is inclusive of the CHA→MC write latency. This provides evidence that unlike C2M-Write domain, P2M-Write domain includes the MC. To determine the P2M-Write domain credits, we run P2M-Write workload from §2.2.1 (which saturates PCIe bandwidth), and apply maximum possible C2M load. We find that IIO write buffer occupancy saturates at ~92, giving us the size of the IIO write buffer.

**P2M-Read.** The P2M-Read domain spans all hops from IIO to DRAM because PCIe reads are non-posted transactions [168]—the IIO needs to wait until reads are serviced from DRAM and data is returned before issuing PCIe completion and replenishing the credits. We were not able to measure IIO read buffer occupancy on our server (and consequently IIO read latency), thus precluding us from determining the precise number of credits and unloaded latency of the P2M Read domain. However, we obtain a lower bound on the P2M-Read

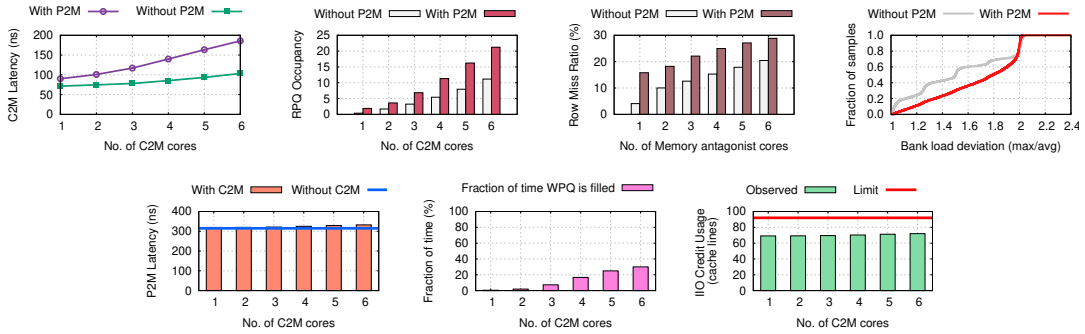



Figure 2.9: Results for understanding quadrant 1 (a-d, top/left-right and e-g, bottom/left-right). All y-axis values are on average. Discussion in §2.5.1.

domain credits using measurements at the CHA (since the number of in-flight P2M-Read requests at the CHA cannot exceed the P2M-Read domain credits). By introducing C2M load colocated with P2M-Read traffic, we found that the number of in-flight P2M-Read requests at the CHA saturates at ~164 cachelines, providing evidence that the P2M-Read domain has a larger number of credits than the P2M-Write domain.

## 2.5 Understanding contention regimes

We now provide an in-depth explanation for the two regimes observed in §2.2 using the lens of domain-by-domain credit-based flow control. We use the same measurement methodology as in §2.4.2. We observe no statistically noticeable change in application performance when counter sampling is enabled. We disable dynamic scaling of core frequency to avoid variation in measurements (we observe less than 1.5% variation across all runs for all counters).

### 2.5.1 Understanding the blue regime

We first focus on quadrant 1 (C2M-Read, P2M-Write) since it captures most of the takeaways in terms of explaining the blue regime. For quadrant 1, we first explain why C2M throughput degrades and then why P2M throughput does not degrade.

**C2M throughput degrades because domain credits are fully utilized and domain latency increases.** Even when the C2M workload is run in isolation, the corresponding domain credits are fully utilized. This is because each core can issue instructions fast enough to keep the LFB full (e.g., a core with 3GHz frequency can issue instructions every 0.3ns, which is more than two orders of magnitude smaller than the minimum C2M-Read domain latency). As a result, any non-zero increase in domain latency will result in throughput degradation. Indeed, when the P2M workload is colocated, we observe $1.26-1.8\times$ increase in domain latency (Figure 2.9(a)) due to queueing at the MC (Figure 2.9(b)) since DRAM is part of the C2M-Read domain. Interestingly, such queueing happens far before memory bandwidth is saturated; we now explain this phenomenon.

Queueing at the MC before memory bandwidth saturation happens due to a combination of two DRAM-level factors: (1) row misses and (2) load imbalance across banks. Row misses result in processing delays at the banks (due to precharge/activate operations), which must be completed before the data can be accessed and transmitted over the memory channel. Even for a workload with 100% row miss ratio, the bank-level processing delays can still be hidden behind data transmission over the memory channel, if requests are load balanced perfectly across the banks. If requests are perfectly distributed across $N_b$

banks, then the bank processing delay can be hidden/overlapped behind transmission on the channel if $t_{\text{Proc}}/N_b < t_{\text{Trans}}$, where $t_{\text{Proc}}$ is the per-request bank processing delay and $t_{\text{Trans}}$ is the per-request transmission delay over the memory channel. This condition holds for the DRAM modules in our setup, where $t_{Proc} \approx 45ns$, $N_b = 32$, and $t_{Trans} = 2.73ns$. In reality, however, load balancing is far from perfect since memory addresses are mapped to banks through a static hash function [177], which does not guarantee perfect load balancing [242]. As a result, requests can be blocked on bank processing even when the memory channel is idle, thus causing queueing even when channel capacity (i.e., memory bandwidth) is not saturated. We quantify row misses and load imbalance, focusing on the single core C2M case in quadrant 1 next.

In the absence of P2M traffic, the row miss ratio for C2M-Read requests is very low (< 4%, shown in Figure 2.9(c)). This is because of the sequential access pattern resulting in a good row locality. Colocating the P2M workload causes a significant increase in row miss ratio (up to 4×). This is because the C2M and P2M workloads access different address spaces — intermixing them reduces row locality, leading to a higher row miss ratio. While row miss ratio also increases for C2M-only traffic from multiple cores since each core accesses a different address space, colocating P2M traffic leads to a larger increase in row miss ratio as is evident in Figure 2.9(c).

To measure load distribution, we sample the number of read requests mapped to each individual bank every 1000 requests[3]. Let the *bank deviation* of a given sample be the ratio of the load of the maximally loaded bank to the average load across banks. Figure 2.9(d) shows the CDF of bank deviation across

[3]For these measurements, we use a dedicated core that busy polls on MC hardware counters [100]. Given constraints on the number of available hardware counters, we focus on 4 banks within a single DRAM module.

10000 samples. We see significant load imbalance, both with and without P2M traffic — the bank deviation is $\geq 1.5\times$ in $50-70\%$ of samples and $\geq 2\times$ in as many as $13-22\%$ of samples (although there is load imbalance even when C2M is run in isolation, it is not a problem since the row miss ratio is very low causing bank processing delays to be negligible).

Returning to our main discussion, P2M traffic observes different behavior compared to C2M traffic in quadrant 1 due to differences in: (1) domain latency and (2) domain credits.

**Write domain does not include execution latency of DRAM, leading to smaller latency inflation relative to reads.** Unlike the C2M-Read domain, where queueing at the MC leads to domain latency inflation, since the P2M-Write domain does not include traversing DRAM, its domain latency only increases when the MC write queue becomes full. As shown in Figure 2.9(f), the fraction of time the WPQ is filled is near zero when P2M-Write is colocated with a single C2M-Read core. Thus, there is no domain latency inflation for P2M-Write (Figure 2.9(e)). With increasing C2M cores, while we see a small increase (< 25ns) in domain latency for P2M-Write (Figure 2.9(e)), it is smaller than what is seen for C2M-Read. This is because the WPQ starts to get filled up occasionally (< 30% of the time), as shown in Figure 2.9(f).

**P2M domain can tolerate latency inflation due to availability of spare domain credits.** Increased P2M-Write domain latency, however, does not lead to a reduction in P2M throughput. This is because, unlike C2M-Read, when the P2M workload is run in isolation, domain credits are not fully utilized (as described in §2.4, unloaded P2M-Write domain latency is ~300ns; therefore, to saturate PCIe bandwidth of ~14GB/s, ~65 credits are needed which is smaller than the

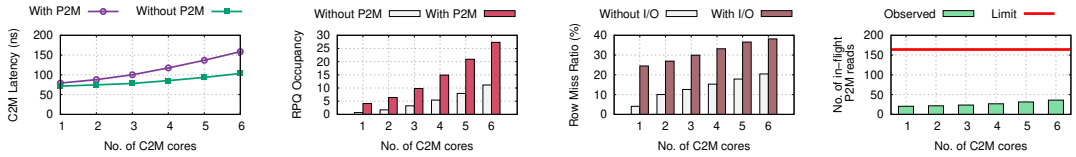



Figure 2.10: Results for understanding quadrant 2 (a-d, left-right). All y-axis values are on average.

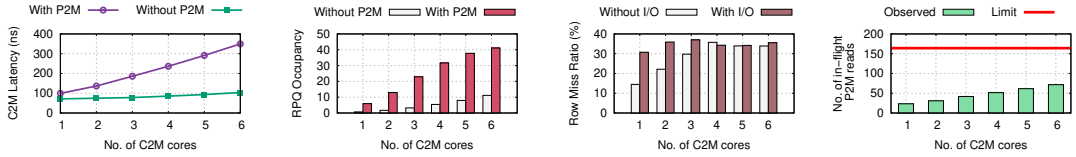



Figure 2.11: Results for understanding quadrant 4 (a-d, left-right). All y-axis values are on average.

~92 available credits). P2M throughput degrades only after domain credits are exhausted. Indeed, we see a slight increase in domain credit utilization with increasing C2M cores, but it is well below the maximum limit (Figure 2.9(g)). Therefore, the P2M-Write domain is able to maintain enough in-flight write requests to mask the latency inflation and thus avoid throughput degradation.

Our explanation of quadrant 1 generalizes to quadrants 2 and 4. The corresponding measurements are shown in Figure 2.10, and Figure 2.11.

In quadrant 2, similar to quadrant 1, we observe C2M throughput degradation before memory bandwidth saturation. This is again due to a combination of row miss ratio increase (Figure 2.10(c)) and bank load imbalance leading to queueing at the memory controller (Figure 2.10(b)), causing domain latency inflation (Figure 2.10(a)) and ultimately C2M throughput degradation. P2M traffic in Quadrant 2 performs reads, and thus observes latency inflation due to queueing at the memory controller, similar to C2M (since the P2M-Read domain includes DRAM). Yet, its throughput does not degrade because there are enough spare domain credits available that allow it to tolerate inflation in do-

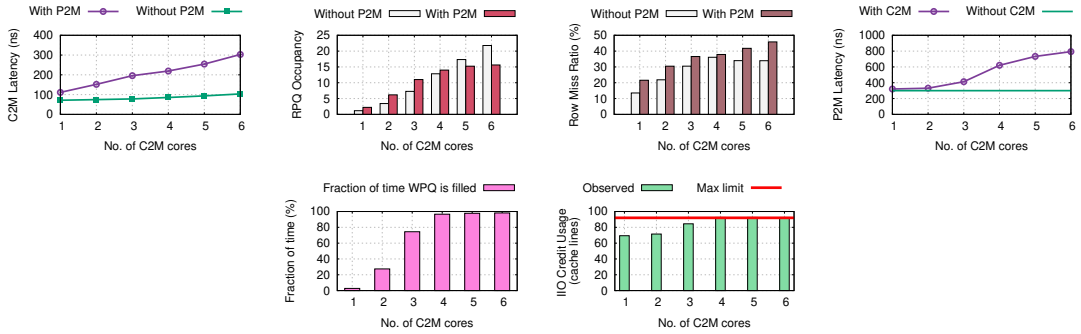



Figure 2.12: Results for understanding quadrant 3 (a-d, top/left-right and e-f, bottom/left-right). All y-axis values are on average. Discussion in §2.5.2.

main latency. Moreover, the spare credit capacity is even larger for the P2M read domain (§2.4)—we observe that the number of in-flight P2M reads[4] is far below this limit even with the maximum number of colocated C2M cores (Figure 2.10(d)).

In quadrant 4, C2M throughput degrades even though memory bandwidth is not saturated due to the same reason as in Quadrants 1 and 2—latency inflation (Figure 2.11(a)) due to queueing (Figure 2.10(b)) caused by a combination of an increase in row misses (Figure 2.11(c)) and load imbalance across banks. P2M throughput does not degrade for the same reason as in Quadrant 2—there are enough spare domain credits to tolerate the domain latency inflation—the measured number of in-flight P2M reads is well below the credit limit even with the maximum number of C2M cores (Figure 2.11(d)).

[4]We were not able to directly measure the IIO read buffer occupancy. Instead, we measure the number of in-flight P2M read requests at the CHA which is a lower-bound on the total number of in-flight requests in the P2M Read domain.

### 2.5.2 Understanding the red regime

We now turn our attention to quadrant 3, where both C2M and P2M observe throughput degradation. Before diving deep, we first briefly revisit the observations in quadrant 3 bearing similarities to §2.5.1.

With 2 or fewer C2M cores, similar to quadrant 1, C2M throughput degrades even before memory bandwidth is saturated (Figure 2.3). We see a similar trend of increase in row miss ratio when P2M is colocated with C2M (Figure 2.12(c)); this, in combination with load imbalance across banks, results in queueing at MC before memory bandwidth is saturated (Figure 2.12(b)). While there is a small (~20 – 30ns) inflation in P2M-Write domain latency (Figure 2.12(d)), akin to quadrant 1, there is no P2M throughput degradation due to spare domain credits (Figure 2.12(f)).

With 3 or more C2M cores, when memory bandwidth becomes saturated, we observe two new trends in this quadrant (§2.2.1). First, from 3 to 4 C2M cores, we observe C2M antagonize P2M (i.e., reduction in C2M throughput degradation coupled with a large increase in P2M throughput degradation). Second, beyond 4 C2M cores, the rate of degradation of P2M throughput reduces with increasing C2M cores. We now discuss the underlying reasons for both observations.

**Backpressure from MC impacts P2M-Write domain but not C2M-Write domain, leading to throughput degradation for the former.** When memory bandwidth is saturated, unlike in quadrant 1 (Figure 2.9(f)), the MC write queue gets filled up persistently (75% of the time with 3 C2M cores, and nearly all the time with 4 or more C2M cores; Figure 2.12(e)), thus leading to backpressure and

causing a backlog of writes at the CHA. Interestingly, this only impacts the P2M workload, but not the C2M workload, even though both are performing writes. This is because the P2M-Write domain spans the MC while the C2M-Write domain does not (§2.4); therefore, backpressure from the MC results in domain latency inflation of the P2M-Write domain but not C2M-Write domain. From 3 to 4 C2M cores, due to backlogging of writes, P2M-Write domain latency increases by 1.5× (Figure 2.12(d)) and results in significant P2M throughput degradation since the domain credits are fully utilized (Figure 2.12(f)). The C2M workload (C2M-ReadWrite), however, is only bound by C2M-Read domain latency (as C2M-Write domain latency does not increase) which only increases by ~12% (Figure 2.12(a)) since reads are not impacted by write backlogging as they can be processed concurrently at the CHA even when writes are blocked. As a result, C2M's share of memory bandwidth increases while P2M's share reduces.

**Backpressure from CHA impacts both C2M and P2M domains, leading to increased degradation for both.** As the write backlog at the CHA continues to increase with increasing C2M load, we observe a new phenomenon that is evident beyond 4 C2M cores: CHA begins to apply backpressure due to limited buffering resources. The trend in Figure 2.12(b) provides evidence of this—the average RPQ occupancy saturates beyond 4 C2M cores (and is lower than the corresponding without P2M values), showing that the number of in-flight read requests from the CHA to the MC has been capped despite the total number of in-flight read requests increasing with more C2M cores. This indicates that some requests are getting blocked at the cores even before being admitted into the CHA—a result of backpressure from the CHA. Under CHA backpressure, write backlogging no longer has a one-sided impact on the P2M domain. Latency inflation is now primarily determined by delay in admitting requests into

the CHA itself, which impacts both C2M and P2M domains. We see a roughly equitable increase in domain latency (~50ns) when going from 5 – 6 cores for both the C2M and P2M domains, thus leading to a relative stabilization of their memory bandwidth shares (Figure 2.3).

## 2.6 Quantitative validation

In §2.5, we identified the root causes for performance degradation due to interplay between processor, memory, and peripheral interconnects, based on correlations with measurements from nodes in the host network. This, however, does not imply that these are the only factors impacting performance degradation. To close this gap and further validate our understanding, we now connect these measurements to the observed end-to-end throughput degradation. To this end, we develop an analytical formula that captures the average memory access latency observed by C2M/P2M traffic (which then directly connects to throughput using Little's law). Our analytical formula focuses on queueing delay at the MC (for reads) and at the CHA (for writes). While, in theory, there can be queueing delay at other points in the host network (e.g. in cache controllers, within the processor interconnect, etc.), we demonstrate that queueing delay at these other points contributes minimally to end-to-end latency (our analytical analysis captures end-to-end performance to a high degree of accuracy across all evaluated workloads). We first describe our analytical formula, following which, we present results of applying it to the four quadrants.

In designing our analytical formula, we exploit the insight that since we are analyzing latency for the purpose of understanding average throughput, we

can focus on average-case behavior across a large number of memory requests rather than on individual request dynamics, which are difficult to capture. Before describing the analytical formula, we highlight that it is not designed to be perfect—it does not capture all the intricacies of DRAM operation, including out-of-order request scheduling, resource contention at some levels of the DRAM hierarchy (e.g., ranks and bank groups), low-level hardware optimizations (e.g., opportunistic processing of memory writes by Intel memory controllers while in read mode [200]), and a subset of DRAM timing constraints (e.g., write recovery delays, rank-level timing constraints, etc.). Despite its relative simplicity, as we will demonstrate, it still captures latency inflation to a reasonable degree of accuracy (within ~ 10% error) in most evaluated scenarios.

We first build the analytical expression for read domain latency following which we discuss write domain latency.

**Read Domain Latency.** Our formula for read domain latency (Figure 2.13) is applicable to both the C2M-Read and P2M-Read domains. As motivated at the start of the section, we focus only on the average queueing delay for reads ($QD_{\text{read}}$) at the MC. Therefore, we abstract away all latency in the end-to-end datapath other than $QD_{\text{read}}$ into a constant ($\text{Constant}_{\text{read}}$). Naturally, $\text{Constant}_{\text{read}}$ is different for the C2M-Read and P2M-Read domains since they have non-shared hops in their datapaths (§2.4). $QD_{\text{read}}$ is expressed as a sum of four additive components. The first three components capture the average delay for a given read request to reach the top of the RPQ (thus, they are all a function of the average RPQ occupancy $O_{\text{RPQ}}$), and the last component captures the additional delay that a request incurs after reaching the top of the queue before it is issued to DRAM. We now describe each of the individual components:

- Switching Delay: This component captures the average time a given read is blocked due to write-to-read switching delay ($t_{\text{WTR}}$). Since we focus on average case behavior, we can compute the total switching cost over a large number of switches (#switches) and average it over a large number of reads ($\text{lines}_{\text{read}}$).
- Write Head-of-Line Blocking: This component captures the average time for which a read is blocked because the channel is currently in write mode and cannot issue reads. Average case analysis allows us to compute the total time spent in write mode over a long time window ($\text{lines}_{\text{written}} \times t_{\text{Trans}}$) and average it across a large number of reads ($\text{lines}_{\text{read}}$).
- Read Head-of-Line Blocking: A given read request has to wait for the requests before it ($O_{\text{RPQ}} - 1$ on average) in the RPQ to get transmitted on the memory channel. In reality, while the MC may schedule requests out-of-order to maximize utilization, our evaluation of the formula indicates that this has very little impact, if any, on the end-to-end latency for the workloads we focus on.
- Top-of-queue delay: Even after reaching the top of the RPQ, a request might still have to wait for activate/precharge operations to complete before it is issued to DRAM. To capture this, we compute the total cost of activate and precharge operations ($\#\text{ACT}_{\text{read}} \times t_{\text{ACT}}$ and $\#\text{PRE}^{\text{conflict}}_{\text{read}} \times t_{\text{PRE}}$) and average them across a large window of requests ($\text{lines}_{\text{read}}$).

**Write Domain Latency.** Writes require slightly different analysis than reads, since writes do not have to wait until they are processed to DRAM. For the P2M-Write domain, they are completed upon admission into the MC WPQ. Thus, P2M-Write domain latency only inflates when the WPQ is filled, at which point requests will have to wait until they are admitted (admission delay $AD_{\text{write}}$).

| Analytical Formula Inputs | |
|---|---|
| $P_{\text{fill}}^{\text{WPQ}}$ | Probability that WPQ is full |
| $N_{\text{waiting}}$ | # write requests awaiting WPQ admission |
| #switches | # switches between read and write mode |
| $\text{lines}_{\text{read/write}}$ | # cachelines read / written |
| $O_{\text{RPQ}}$ | Average RPQ occupancy |
| $\text{PRE}_{\text{read/write}}^{\text{conflict}}$ | # precharges due to row conflicts for reads / writes |
| $\text{ACT}_{\text{read/write}}$ | # activations for reads / writes |

Table 2.2: Inputs to the formula for computing latencies for C2M and P2M read/write domains.

$$
\begin{aligned}
L_m^{\text{read}} &= \text{Constant}_{\text{read}} + QD_{\text{read}} && \text{(Average read latency)} \\
QD_{\text{read}} &= O_{\text{RPQ}} \cdot \frac{\#\text{switches}}{\text{lines}_{\text{read}}} \cdot t_{\text{WTR}} && \text{(Switching Delay)} \\
&+ O_{\text{RPQ}} \cdot \frac{\text{lines}_{\text{written}}}{\text{lines}_{\text{read}}} \cdot t_{\text{Trans}} && \text{(Write HoL blocking)} \\
&+ (O_{\text{RPQ}} - 1) \cdot t_{\text{Trans}} && \text{(Read HoL blocking)} \\
&+ \frac{\#\text{ACT}_{\text{read}}}{\text{lines}_{\text{read}}} \cdot t_{\text{ACT}} + \frac{\#\text{PRE}_{\text{read}}^{\text{conflict}}}{\text{lines}_{\text{read}}} \cdot t_{\text{PRE}} && \text{(Top-of-queue delay)}
\end{aligned}
$$

Figure 2.13: Read domain latency components (inputs defined in Table 2.2; $t_{\text{WTR}}$, $t_{\text{Trans}}$ (transmission delay: time taken to transmit a single cacheline over the memory channel in either direction), $t_{\text{ACT}}$ ($t_{\text{RCD}}$) and $t_{\text{PRE}}$ ($t_{\text{RP}}$) are standard DRAM timing constraints).

$$
\begin{aligned}
L_m^{\text{write}} &= \text{Constant}_{\text{write}} + AD_{\text{write}} && \text{(Average write latency)} \\
AD_{\text{write}} &= P_{\text{fill}}^{\text{WPQ}} \cdot X_{\text{write}} \\
X_{\text{write}} &= N_{\text{waiting}} \cdot \frac{\#\text{switches}}{\text{lines}_{\text{written}}} \cdot t_{\text{RTW}} && \text{(Switching Delay)} \\
&+ N_{\text{waiting}} \cdot \frac{\text{lines}_{\text{read}}}{\text{lines}_{\text{written}}} \cdot t_{\text{Trans}} && \text{(Read HoL blocking)} \\
&+ (N_{\text{waiting}} - 1) \cdot t_{\text{Trans}} && \text{(Write HoL blocking)} \\
&+ \frac{\#\text{ACT}_{\text{write}}}{\text{lines}_{\text{written}}} \cdot t_{\text{ACT}} + \frac{\#\text{PRE}_{\text{write}}^{\text{conflict}}}{\text{lines}_{\text{written}}} \cdot t_{\text{PRE}} && \text{(Top-of-queue delay)}
\end{aligned}
$$

Figure 2.14: Write domain latency components (inputs defined in Table 2.2; $t_{\text{WTR}}$, $t_{\text{Trans}}$ (transmission delay: time taken to transmit a single cacheline over the memory channel in either direction), $t_{\text{ACT}}$ ($t_{\text{RCD}}$) and $t_{\text{PRE}}$ ($t_{\text{RP}}$) are standard DRAM timing constraints).

Our write domain latency formula (Figure 2.14) captures $AD_{\text{write}}$ via (1) the probability that a request is blocked due to the WPQ being full ($P^{\text{fill}}_{\text{WPQ}}$) and (2) the average waiting time for a request when the WPQ is full ($X_{\text{write}}$). For $X_{\text{write}}$, we use an expression analogous to read queueing delay, with the parameters for reads/writes swapped (since the corresponding components for write processing are exactly the duals of those for reads), and using $N_{\text{waiting}}$, the average number of writes (both C2M and P2M) waiting to be admitted into the queue, instead of $O_{\text{RPQ}}$ (since admitting $N_{\text{waiting}}$ requests requires processing an equal number of writes to make space in the queue). Unlike P2M writes, C2M writes do not have to wait until they are admitted to the MC. We do not capture inflation of C2M-Write domain latency and assume it to be a constant. We later discuss the implications of doing so.

All formula inputs can be captured or derived using programmable uncore performance counters available on Intel servers [100] using the same measurement methodology as §2.5. We use MC counters to capture all the inputs except $N_{waiting}$. For $N_{waiting}$, we use counters from the CHA, since this is where requests are backlogged when the MC write queues are full [100].

**Applying the formula.** We set $Constant_{read/write}$ based on unloaded latencies of domains (§2.4.2). For any given experiment, we then apply the formula using the measured inputs to obtain the average domain latency. Depending on the workload, we use either the read or the write domain latency expression. For C2M/P2M-Read, we use the read latency expression. For P2M-Write, we use the write latency expression. For C2M-ReadWrite, we use the C2M-Read latency plus a constant (to account for C2M-Write). After obtaining average domain latency ($L$), we compute estimated throughput using the expression in §2.4.

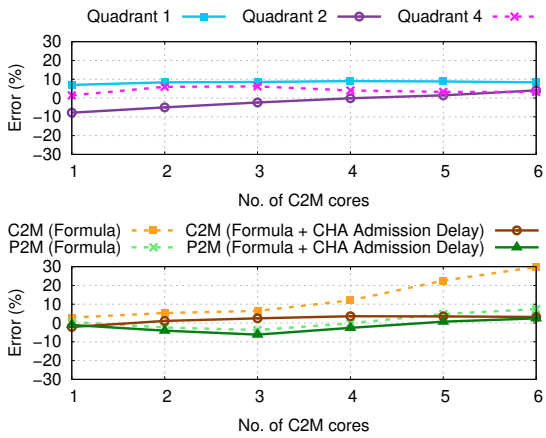



Figure 2.15: Accuracy of the analytical formulae: (top) Error in the formula's estimate of C2M throughput for quadrants 1, 2, 4. (bottom) Error in the formula's estimate of C2M and P2M throughput for quadrant 3 (both with/without adding CHA admission delay). Positive values indicate overestimation, and negative values indicate underestimation.

**Formula accurately captures end-to-end throughput.** Figure 2.15 shows the error in the formula's estimate of throughput in all the quadrants from §2.2.1. The formula captures throughput for the C2M workload (which is what degrades) within 10% error for all data points in quadrants 1, 2, and 4. For quadrant 3, with 4 or fewer cores, the error for both C2M/P2M is within 10%. However, beyond 4 C2M cores, the error increases (especially for C2M throughput). This is because our formula currently does not account for admission delay to the CHA, which manifests when CHA buffers get filled up as is the case for quadrant 3 with 4 or more C2M cores and causes latency inflation for both C2M and P2M domains (§2.5.2)—as a result, the formula underestimates latency leading to overestimation of throughput. When we add the measured CHA admission delay from the testbed to the output of the formula, the error reduces to $< 10\%$ for all data points, thus validating our understanding.

**Breakdown of formula components.** Figure 2.16 presents the breakdown of queueing delay into each of the individual formula components for all of the

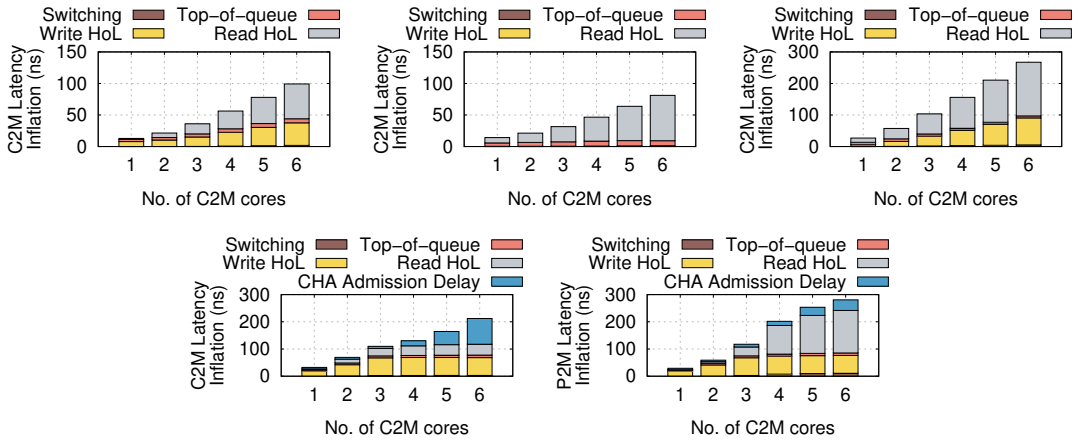



Figure 2.16: Breakdown of analytical formula components (a-c, top/left-right, d-e, bottom/left-right) (a, b, c) Breakdown of formula components for C2M in quadrants 1, 2, and 4 respectively; (d, e) Breakdown of formula components for C2M and P2M along with CHA admission delay in quadrant 3.

quadrants. For quadrant 1, with a single C2M core, WriteHoL is the dominant contributor. With increasing C2M cores, both WriteHoL and ReadHoL increase. Quadrant 2 has a larger Read HoL component due to higher average read queue occupancy, but it has no Write HoL component because there are no writes. In quadrant 4, ReadHoL is the dominant contributor for all data points. In quadrant 3, for C2M, WriteHoL is the dominant contributor up to 4 C2M cores, beyond which CHA admission delay starts to dominate. For P2M, WriteHoL is the dominant factor until 3 C2M cores, after which ReadHoL becomes dominant.

## 2.7 Related work

We discuss the most closely related prior works in the context of our study.

**C2M/P2M Contention.** Recent works from Google [3] and Alibaba [138] have reported the impact of C2M/P2M contention. The common observation in both

of these works is that when host memory bandwidth is bottlenecked, C2M antagonizes P2M bandwidth. Our characterization captures this performance degradation regime, and our analysis explains the core underlying microarchitectural reason for this behavior. DDMA [131] considers the C2M/P2M contention problem with great prescience. However, it does not provide an in-depth study of the problem and does not capture the trends observed recently in datacenters. The authors present an interesting architecture to isolate C2M and P2M traffic. However, it requires specialized Dual Data-Port DRAM which is not deployed in datacenters today, thus limiting its practical applicability.

**Multi-core Memory Scheduling.** Our work is complementary to the large body of literature on memory scheduling for multi-core architectures [64, 80, 108, 122, 123, 158, 159, 161–163, 166, 205–207] which focus on C2M-only traffic. It would be interesting to explore extensions of these works while taking P2M traffic into account. Intel MBA [90] enables controlling memory bandwidth allocation for applications running on CPU cores, but does not take P2M traffic into account.

**Memory scheduling for heterogeneous systems.** There have been some works on memory scheduling for heterogeneous architectures [21, 112, 117]. These works primarily focus on the setting of CPU cores and GPUs integrated into the same chip. Although these works target a different context than ours, their ideas and mechanisms could be leveraged for better scheduling of C2M/P2M requests in the datacenter context. For instance, it may be possible to leverage the scheduler architecture proposed in [21] to efficiently implement prioritization policies between C2M/P2M traffic.

**DRAM Latency Analysis.** Prior work [40, 41] extensively analyzes DRAM ac-

cess latency when there is no contention. Our work is focused on analyzing memory access latency when there is C2M/P2M contention in order to explain the observed throughput degradation trends. There has been a lot of work on modeling DRAM behavior. While most of these works [6, 46, 224, 236] focus on capturing saturation bandwidth, ANATOMY [88] proposes a queueing theoretic model for DRAM latency. Our work is orthogonal to these efforts — we do not try to analytically model DRAM — instead, our analytical formula (§2.5) simply connects low-level measurements on the testbed to the observed latency inflation.

**Cache contention and DDIO.** Since we focus on memory contention, our work is complementary to prior work that addresses LLC contention between CPU cores [107, 108, 116, 120, 167, 169], works that analyze DDIO [35, 67, 226], works that optimize DDIO usage for specific use cases [66,137,149,210], and works that propose DDIO improvements [9]. Recent work [237] which studies the impact of LLC contention between CPU cores and Peripheral devices is the closest to our study in this domain. The observations in this work are complementary to ours. For instance, it shows that the performance of LLC-bound workloads running on CPU cores can be significantly impacted by Peripheral traffic with DDIO due to contention for LLC capacity. We show that even for non LLC-bound workloads, DDIO can amplify performance degradation due to memory contention.

## 2.8 Implications for the rest of the thesis

We have presented a conceptual abstraction of domain-by-domain credit-based flow control that precisely captures the interplay between processor, memory, and peripheral interconnects within the host network. Using this abstraction, we have built an in-depth understanding of contention within the host network and its impact on application performance reported by previous studies, as well as identified new, previously unreported, regimes of contention within the host network.

Chapter 3 builds on the measurement methodology introduced in this chapter to extend understanding of the host network to more recent processors with chiplet-based architectures and cache-coherent interconnects for memory disaggregation such as CXL. Our analysis in this chapter revealed that memory interconnect contention can impact application performance far before memory bandwidth is saturated (§2.2), due to memory access latency inflation (§2.5). Chapter 4 builds on this observation to propose memory access latency as a unifying lens for OS memory management over cache-coherent interconnects. In doing so, Chapter 4 leverages the relationship between memory access throughput and memory access latency enabled by the flow-control lens (§2.5), to establish a principle for data placement across local and disaggregated memory. Furthermore, Chapter 4 also builds on the understanding of host architecture and measurement methodology introduced in this chapter to design an efficient and accurate mechanism for measuring the real-time access latencies of local and disaggregated memories.

CHAPTER 3

# UNDERSTANDING CACHE-COHERENT INTERCONNECTS FOR MEMORY DISAGGREGATION

In this chapter, we first extend the background on host architecture and interconnects provided in Chapter 2, to include chiplet-based architectures and cache-coherent interconnects for memory disaggregation such as CXL (§3.2). Next, we build an in-depth understanding of CXL latency (§3.3) and bandwidth (§3.4) properties. We then characterize the impact of data placement across local and (CXL-exposed) disaggregated memory on application performance (§3.5). Finally, we discuss related work (§3.6) and the implications of this chapter to the rest of the thesis (§3.7).

## 3.1 Overview

The feasibility of memory disaggregation critically depends on its impact on application performance. While memory disaggregation over cache-coherent interconnects minimizes overhead by obviating the need for software-level interposition in the critical path (§1), the optimal placement of data across local and disaggregated memory and the resulting application performance depend on the latency and bandwidth characteristics of the cache-coherent interconnect used for memory disaggregation (relative to the latency and bandwidth characteristics of the traditional DDR memory interconnect).

CXL has emerged as the de facto standard cache-coherent interconnect for memory disaggregation on CPUs. It is supported on all recent Intel and AMD processors, and has been deployed at scale in both Microsoft Azure [156] and

Meta [79]. Prior works [57, 79, 141, 208] have extensively characterized latency and bandwidth characteristics of real CXL hardware and their impact on application performance. However, they lack an in-depth understanding of the root causes of observed CXL latency overheads and/or bandwidth inefficiencies:

- In recent processors with state-of-the-art CXL devices, the end-to-end unloaded latency of CXL memory accesses has been reported to be ~2× higher than that of local DDR memory access latency (adding ~100 – 130ns of additional latency) [57, 79, 141, 208, 246]. Conceptually, CXL is expected to have higher latency due to it being a serial interconnect as opposed to a parallel interconnect like DDR. However, this does not satisfactorily explain the observed latency overheads in real CXL hardware. First, electrical models and hardware IP measurements [194] suggest that CXL (de-)serialization incurs a round-trip latency overhead of 42 – 50ns which is significantly smaller than the end-to-end CXL latency overhead observed in practice. Second, serial memory interconnects such as OMI [54] have reported end-to-end memory access latency overheads as small as 10ns [45], which begs the question of how fundamental CXL latency overheads are.
- Recent work [141] reports that CXL introduces higher and "unstable" tail memory access latencies. While the authors speculate that this may be caused by CXL flow control, thermal management or memory controller inefficiencies, the underlying root cause remains unknown.
- Many prior CXL bandwidth measurements use either only a small subset of available PCIe lanes on the processor [141], not fully mature CXL memory controller implementations [208] or older generation DDR memory channels and modules [79]. This leaves a gap in our understanding of CXL's peak memory bandwidth capabilities.

Building systems software and hardware for memory disaggregation in the absence of an in-depth understanding of CXL performance is far from ideal because it risks misinformed and shortsighted design decisions. For example, despite the lack of in-depth understanding of CXL latency overheads, some have gone as far as to use this as a reason to argue against the feasibility of CXL-based memory disaggregation altogether [135].

In this chapter, we advance the status-quo of CXL performance understanding. We characterize the latency and bandwidth of CXL memory accesses in a recent processor (Intel Granite Rapids) with two commercially available state-of-the-art CXL devices. What differentiates our analysis from prior work is that we measure one level deeper. First, to differentiate between CXL overheads originating in the CXL device or memory controller and those within the processor, we use a PCIe/CXL analyzer that allows us to interpose on CXL memory requests and responses as they traverse the CXL link between the processor and the CXL memory device. Second, we infer a finer-grained understanding of overheads within the CXL memory controller by measuring differences in performance characteristics of requests of different types (read, write, link layer retry) that traverse different subsets of the internal CXL memory controller datapath. Third, to gain further insight into CXL overheads within the processor, we reverse engineer the datapath and routing of CXL memory requests within the processor (between the CPU core issuing the memory request and the processor root complex). In combination, this allows us to obtain finer-grained breakdowns of CXL memory access latency at both the average and tails and characterize CXL bandwidth efficiency. Finally, we connect CXL latency and bandwidth characteristics to application-level performance as a function of data placement across local and CXL-attached disaggregated memory. Overall, our

analysis both corroborates and complements prior observations (by providing deeper understanding) as well as course-corrects and clarifies misconceptions. Our key takeaways are the following:

- **CXL latency overheads are not merely due to (de-)serialization costs, but also due to poor interplay between CXL and existing processor interconnect locality abstractions:** For the CXL device with the smallest average latency (under single in-flight request) that we study, we find that CXL-related overheads within the device account for only 45% of end-to-end CXL latency overhead (31% from serialization, deserialization and protocol processing, 14% from multiplexing requests from CXL ports to DDR channels). The remaining latency overhead originates from within the processor, between the processor core and root complex. We find that more than half of this overhead (30% of end-to-end latency overhead) is due to poor interplay between CXL and existing processor memory locality abstractions. In modern processors, cores and CHA / L3 cache slices are distributed across multiple chiplets. For local DDR memory accesses, the processor allows localizing mappings of physical addresses to CHA / L3 cache slices within the same chiplet. However, for CXL memory accesses, the physical address space of even a single device is scattered across all chiplets; as a result, even if a processor core is physically close to the root complex, a memory request that gets mapped to an L3 cache slice on a faraway chiplet incurs latency overhead of one or more cross-chiplet crossings before it can reach the root complex and traverse the CXL link.
- **CXL-induced tail latencies are a myth:** We find that P99 and P99.9 tail latencies under single in-flight request are not an artifact of CXL, but rather a fundamental property of DRAM itself. Indeed, careful measurements

reveal similar tail latency inflation even for local DDR memory accesses. Using a DRAM analyzer, we further validate that this tail latency inflation (observed for both local DDR and CXL) is because of periodic DRAM refreshes which block memory accesses while they are being processed.

- **Do not underestimate CXL bandwidth capabilities:** We find that read-only CXL memory traffic is able to achieve 81% of the theoretical maximum unidirectional PCIe bandwidth, with the remaining gap being explained by header and flit framing overheads. Read-write traffic over CXL is able to simultaneously utilize both directions of the PCIe link, while over DDR it is limited by being able to transmit in only one direction at any point of time. Although aggregate memory bandwidth over CXL is comparable but smaller than memory bandwidth over DDR in absolute terms, we find that CXL achieves 2.91× and 4.49× higher bandwidth per processor pin for read-only and read-write traffic respectively.
- **Offloading data to CXL-attached disaggregated memory can improve application performance:** Discussions around memory disaggregation often focus on quantifying the degradation in application performance caused by disaggregating memory. We find that offloading data to CXL-exposed disaggregated memory can actually *improve* application performance by up to 78%. This is because, under multiple in-flight requests, DDR access latency can increase by up to 4.8× the DDR latency under single in-flight request, due to interconnect contention, causing it to exceed the minimum CXL access latency by up to 2.14×. In such a regime, offloading data to CXL reduces the load on the DDR memory interconnect—thereby reducing its access latency—and improving application performance by leveraging the additional bandwidth offered by CXL.

## 3.2 Background on emerging host architectures and cache-coherent interconnects

**Brief background on CXL interconnect.** CXL is an industry standard interconnect protocol designed to enable a number of use cases including memory capacity/bandwidth expansion, memory disaggregation and finer-grained communication between processors and accelerators [57]. The protocol is organized into three layers. At the physical layer, CXL uses the PCIe Gen 5 physical layer. The link layer is a custom layer that provides credit-based flow control and reliability. At the transaction layer, different transaction layer protocols serve different use cases; here we focus on CXL.mem, which enables processors to access external memory through standard load and store instructions in a cache-coherent fashion. Furthermore, we focus on CXL devices that do not internally cache data (also called "Type 3" or memory expansion devices). A CXL.mem read transaction consists of a request from host to device (M2S Req) carrying the address, to which the device replies with the requested cacheline data (S2M DRS). A write transaction consists of a request from host to device (M2S RwD) carrying the address and cacheline data, to which the device replies with a completion (S2M NDR) [57]. Requests may also carry an optional metadata value, that the host can ask the device to store alongside the cacheline and return on subsequent requests, which is useful for storing coherence-related state (*e.g.*, directory information).

**Granite Rapids host architecture: Chiplets and CXL integration.** The high-level architecture of a Granite Rapids host, illustrated in Figure 3.1, is similar to that discussed in § 2.3. The key differences are (1) a chiplet-based architecture,

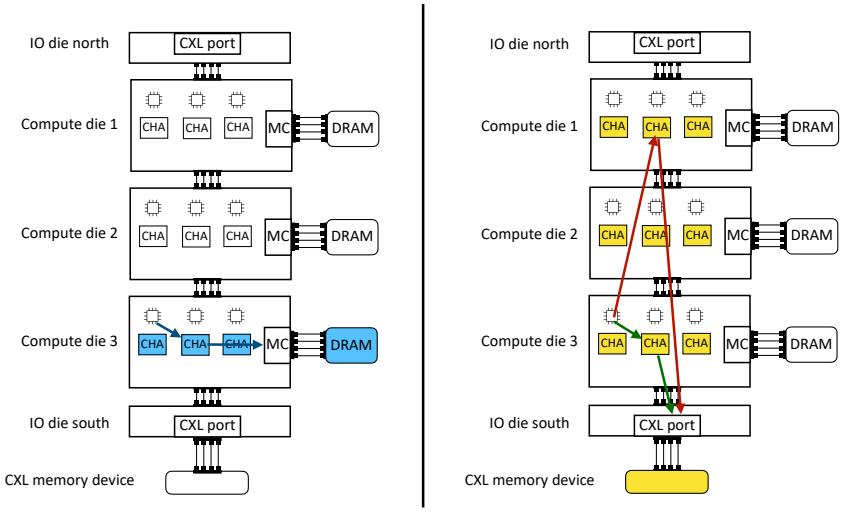



Figure 3.1: Architecture of CXL-enabled Intel Granite Rapids processor and datapath of DDR and CXL memory requests. The figure illustrates a single processor socket drawn twice: (left) shows CHA address mapping and datapath for DDR requests (right) shows CHA address mapping and datapath for CXL requests.

leading to a processor interconnect with deeper topology, and (2) the integration of CXL. Chiplet-based architectures enable scaling to larger core counts by distributing cores across multiple interconnected dies within each socket. In the remainder of our discussion, we use the terms chiplet and die interchangeably. A single socket of a Granite Rapids host has 3 compute dies and 2 IO dies. Each compute die contains processor cores, LLC/CHA slices and DDR memory controllers, all of which are interconnected by an on-die mesh interconnect. Each IO die contains a root complex which connects to PCIe lanes (to connect to traditional peripheral devices and/or CXL devices) as well as inter-socket interconnect lanes and corresponding controllers. Disaggregation of the compute and IO dies enables larger beachfront for IO pins—the IO dies are sized to have smaller area and thus have larger perimeter per-unit area. The compute and IO dies are interconnected sequentially, as shown in the figure, using proprietary inter-chiplet interconnect links. A CXL memory device (a device which exposes memory to the processor over the CXL protocol) can be attached to the root

complex via PCIe lanes; there is a "CXL port" within the root complex which implements the CXL link and transaction layers. Both DDR and CXL memory are mapped into disjoint regions of the host physical address space.[1] The CHA is responsible for routing memory requests based on physical addresses; each CHA slice owns a disjoint subset of physical addresses.

**Datapath of memory requests: DDR versus CXL.** We now describe the datapath of memory read requests as illustrated in Figure 3.1; our focus here is on memory requests originating from processor cores (i.e., C2M requests). A request first goes from the core to a CHA slice, routed to the slice based on its physical address. Depending on the physical address, the CHA then routes it based on whether it maps to DDR or CXL memory. If it is a DDR request, it is forwarded to a DDR memory controller, which processes it as usual and returns the response; the response can be sent directly to the core via an optimization called D2C which is designed to minimize latency (a notification/ack is also sent to the LLC/CHA slice concurrently so that state can be cleared; the coherence protocol takes care of conflicts; we do not know the details). If it is a CXL request, it is forwarded to the root complex on the IO die; it goes through the CXL protocol stack within the root complex and over the CXL/PCIe link to the CXL memory device; it then goes through the CXL protocol stack within the CXL port on the device-side and is forwarded to one of the memory channels, where DRAM access happens and the response is returned; the response goes from the root complex directly to the core (via D2C).

**CHA address mappings.** The mapping of physical addresses to CHA slices is

[1] Intel Granite Rapids processors also support an optional feature called Flat Memory Mode where DDR memory acts as a transparent hardware-managed cache backed by CXL memory [246]. This is outside the scope of our analysis.

done using proprietary hash functions. We reverse-engineered these address mapping functions to understand how DDR and CXL memory requests are routed within the processor (details in §3.3). To enable localized routing of memory requests, the processor supports a feature called Sub-NUMA Clustering (SNC). When this is enabled, addresses corresponding to a given DDR channel are mapped to only CHA slices local to the compute die to which the DDR channel is attached. As illustrated in Figure 3.1, this ensures that DDR memory accesses do not incur any die-to-die crossings, thus minimizing access latency (at the cost of reduced available L3 cache capacity). CXL address mappings, on the other hand, do not interplay well with SNC. The addresses corresponding to a given CXL device are distributed across all CHA slices across all compute dies. Consequently, as illustrated in Figure 3.1, CXL memory requests can incur up to 4 additional die-to-die crossings in the worst-case. This results in higher average memory access latency for CXL requests while also providing higher accessible L3 cache capacity. In the DDR case, one can choose between lower memory access latency or higher L3 cache capacity (by enabling or disabling SNC). However, for CXL, this trade-off is fixed. We discuss the performance implications of this design decision in §3.3.

**Inside a CXL memory controller / device.** Figure 3.2 illustrates a model of the internal architecture of a CXL memory device. The CXL port implements the PCIe PHY and the CXL link and transaction layer processing. An internal interconnect maps CXL requests to memory channels. For DRAM access, assuming the DRAM memory is exposed via DDR, each channel has a memory controller (MC) that implements DRAM access and scheduling; similar to a standard memory controller (§ 2.3), it has separate queues for read and write requests.

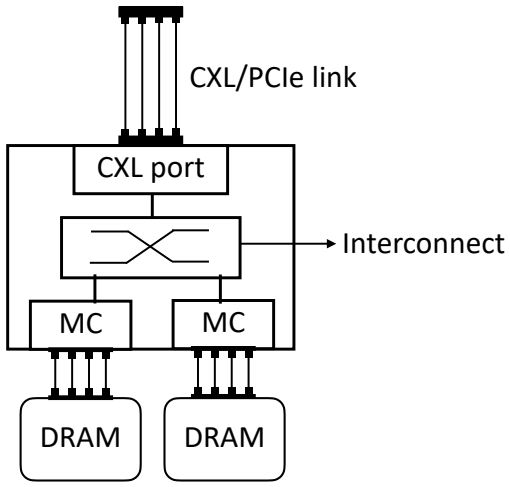



Figure 3.2: Internals of a CXL memory device.

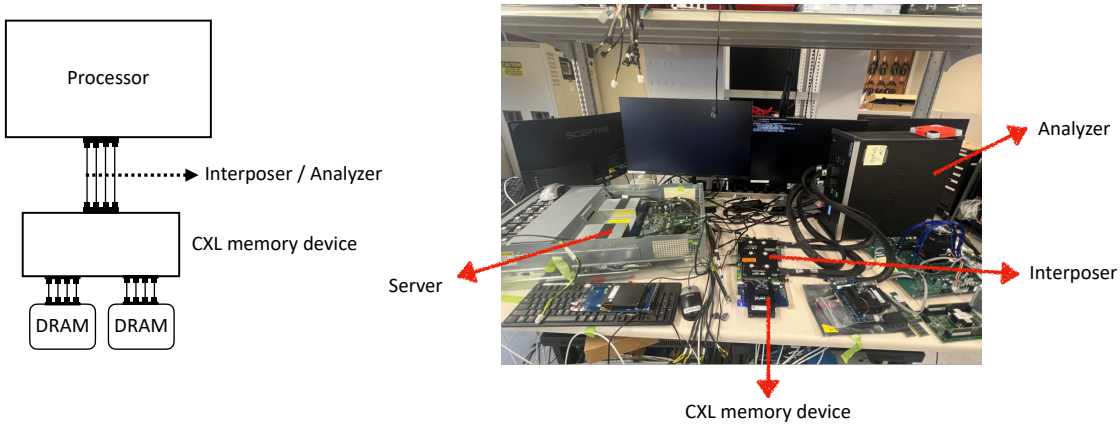



Figure 3.3: Testbed setup for CXL performance measurements.

## 3.3 Understanding latency

Unlike traditional DDR-attached memory, the access latency of CXL even under single in-flight memory request (i.e., the minimum hardware latency) is not well understood. In this section, we study the access latency of memory read requests under single in-flight request over CXL (relative to DDR). We consider both average access latency and tail latency. §3.5 provides insights into access latency under multiple in-flight requests. Before presenting our results, we first discuss our experimental setup and measurement methodology.

**Testbed.** We use a dual socket server with Intel Granite Rapids (GNR-AP) processors. Each socket has 96 cores and 120 CHA slices distributed evenly across 3 compute dies (2.6GHz core frequency and 480MB total L3 cache capacity). Each compute die has 4 DDR5 6400MHz channels with single attached DRAM module. All experiments are run on single socket. Power saving features are turned off to enable deterministic measurement (latency-optimized mode). We use two commercially available CXL memory devices which we refer to as CXL-A and CXL-B. CXL-A has an x16 CXL interface and 2× DDR5 5600MHz channels. CXL-B has an x8 CXL interface and DDR4 memory channels. Unless otherwise specified, we enable SNC to minimize memory access latency.

We use a Teledyne LeCroy T516 PCIe Gen 5 analyzer with CXL support, attached via an interposer, to gain a point of interposition on the CXL datapath. As illustrated in Figure 3.3, the interposer sits between the server and the CXL memory device. It is designed to operate at line rate and remains transparent both in terms of functionality and performance (that is, introducing the interposer changes neither the functional behavior nor the performance of the system). The interposer is connected to the analyzer, which can be programmed to record traces and capture both CXL link and transaction layer packets. The analyzer can be configured to start recording on a specific trigger condition (*e.g.*, request to a specific address or with a specific payload) and records all packets (with configurable filter conditions) until the recording buffer gets filled. One can connect to the analyzer via USB and analyze traces offline through Teledyne's software.

**Reverse-engineering CHA address mapping.** As discussed in §3.2, the mapping of physical addresses to CHA/LLC slices is governed by proprietary, un-

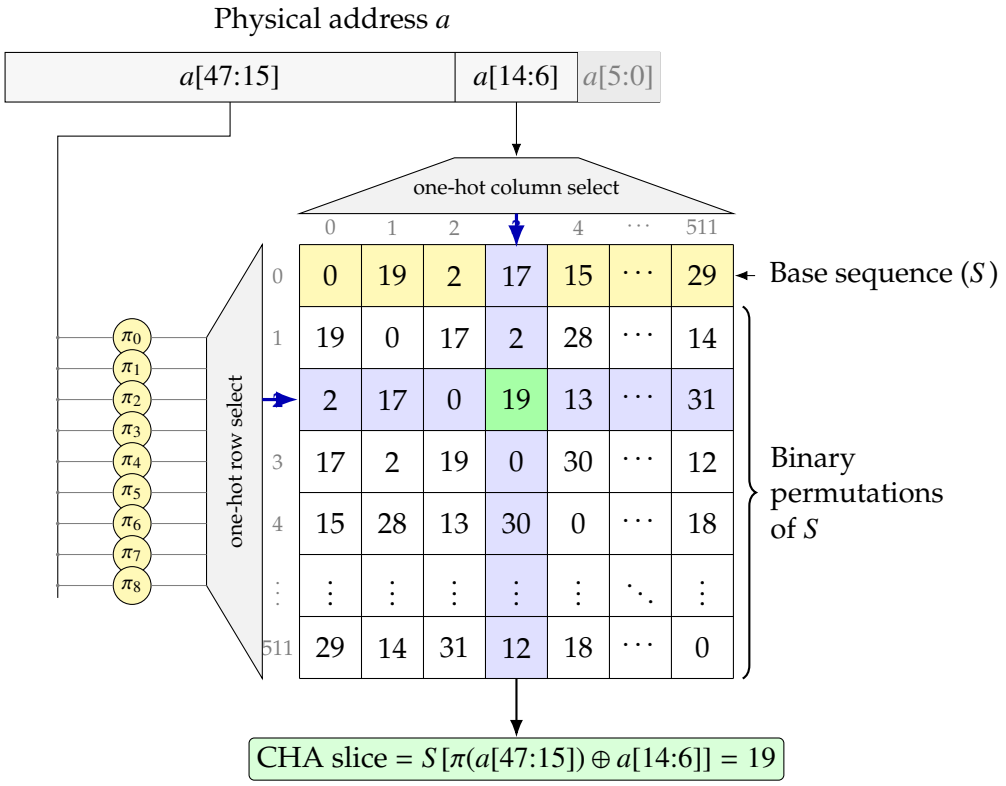



Figure 3.4: Structure of the CHA address mapping function. The mapping can be visualized as a $512 \times 512$ table whose first row is the base sequence $S$ and whose every other row is a binary permutation of it. The 9 low-order bits $a$[14:6] select the column, while 9 XOR-parity functions $\pi_0 \ldots \pi_8$ of the higher-order bits $a$[47:15] select the row; the selected entry is the output CHA slice. Note that, hardware implementations of this function do not necessarily have to materialize the entire two-dimensional table—one can compute the CHA slice directly given the physical address, $\pi$ and $S$ using the expression shown in the green box at the bottom; viewing the function as a two-dimensional table lookup is primarily useful for easy visualization.

documented hash functions. To understand the impact of memory request routing within the processor on access latency, we empirically reverse-engineer the CHA address mapping functions on our Granite Rapids processor, for both DDR- and CXL-backed physical addresses. Before diving into the minutiae, we first briefly summarize our key findings related to the CHA address mapping functions.

For DDR-backed memory, we find that addresses are mapped, at cacheline

granularity, to the 40 CHA slices within the compute die where the DDR channel is attached. For CXL-backed memory, we find that addresses are distributed across all 120 CHA slices across all compute dies with an additional outer layer of interleaving: 1KB-granularity blocks are distributed round-robin across the 3 compute dies, and within each 1KB block, addresses are mapped to CHA slices within the corresponding compute die using a mapping function similar to that used for DDR mappings.

We now describe the details of our reverse-engineering process. We build on observations from prior work [153] on the general structure of Intel's CHA address mapping functions (illustrated in Figure 3.4): the physical address space is divided into blocks, each spanning a fixed power-of-two number ($2^k$) of cache-lines. A fixed *base sequence (S)* of slice numbers, indexed by the low-order cache-line address bits, is repeated across the physical address space with a per-block binary permutation[2] of the index. The permutation applied to each block is determined by $k$ permutation selectors ($\pi_i$ for $i \in 0..k-1$) each of which is a XOR-reduction (parity) of subsets of the higher-order address bits. For example, say we have 40 CHA slices and the block size is 512 ($k = 9$). The CHA slice for a physical address $a$ is computed as slice $= S[\,(a[14:6]) \oplus \pi(a[47:15])\,]$, where $S$ is the 512-entry base sequence (each entry in $\{0, \ldots, 39\}$), the 9-bit index is drawn from cache-line address bits 6–14, and the 9-bit permutation selector $\pi(a)$ has each of its bits equal to the parity of $a$ under a distinct bitmask over address bits 15–47. This computation can be visualized as a table lookup as illustrated in Figure 3.4. Assuming this structure for the mapping function, to reverse-engineer

[2] A binary permutation of a sequence of length $2^k$ is the reordering obtained by XORing the binary index of each element with a fixed $k$-bit constant $p$; i.e., the permuted sequence $S'$ satisfies $S'[i] = S[i \oplus p]$. Since XORing with a constant is a bijection on $\{0, \ldots, 2^k - 1\}$, this is indeed a permutation. A sequence of length $2^k$ has $2^k$ possible binary permutations (one per $k$-bit constant $p$, including the identity).

it, we need to recover the base sequence $S$ and the permutation selectors $\pi_i$. We do so in two steps. First, we collect a dataset of physical address to CHA slice mappings. Second, we find $S$ and $\pi_i$s consistent with the address mappings in the dataset.

We infer the CHA slice of a given physical address using CHA hardware counters that track the number of requests received per CHA slice. For a given address, we repeatedly issue non-temporal store instructions to the target cache-line (ensuring memory requests are generated by bypassing caches) and sample all CHA counters before and after. The CHA slice whose counter increments the most—provided it accounts for a clear majority of the observed transactions—is recorded as the owner of that address. Physical addresses are obtained by allocating 2MB huge pages on the target NUMA node (DDR or CXL) and translating virtual to physical addresses via `/proc/self/pagemap`. To efficiently cover the relevant bits, we sweep all 512 cache lines within each aligned 32KB block (exercising index bits 6–14) for many (1000) blocks chosen at random across a large buffer (randomizing the higher-order bits 15+ that drive the permutation selectors).

Given the dataset, naively brute-forcing over all possible $S$ and $\pi_i$s is not tractable—assuming 40 CHA slices and block size of $2^9 = 512$, there are $40^{512}$ possible base sequences and $2^{33}$ possible bitmasks for each $\pi_i$. To make the problem tractable, we leverage two key observations. First, looking at any block of 512-aligned consecutive-address slice numbers gives us one of the binary permutations of the base sequence $S$. From this we can generate all 512 possible binary permutations of $S$, one of which is the actual base sequence. This reduces the search space of candidate base sequences to just 512. Second, given

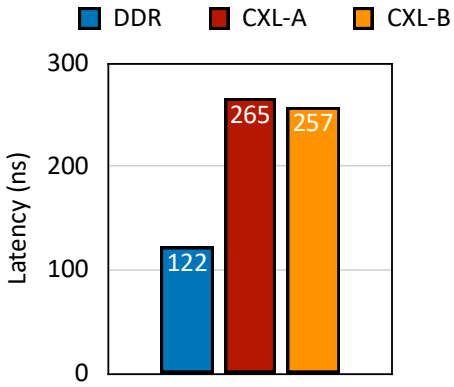



Figure 3.5: Average memory access latency under single in-flight request for DDR and CXL.

a base sequence, one can solve for the permutation selectors ($\pi_i$) in polynomial time: each of the 9 bits of $\pi$ is (by construction) a linear function over GF(2) of the higher-order address bits, we set up a system of linear equations over GF(2)—one equation per sampled block in the dataset, relating address bits 15–47 to the observed permutation bit—and solve it via Gaussian elimination to obtain the bitmask for each $\pi_i$. We verify the recovered function by checking that it exactly reproduces the measured CHA slice for every address in a held-out dataset.

### 3.3.1 Average latency

We use a simple pointer chasing benchmark (with prefetchers disabled) to characterize average latency under a single in-flight request. Figure 3.5 shows the results for DDR, CXL-A and CXL-B. Interestingly, the DDR access latency observed on this server is larger than the latency observed in the older-generation processors used in Chapter 2[3]. CXL-A and CXL-B have 2.17× and 2.10× higher latency than local DDR. The absolute latency gap between CXL and local DDR

[3]We are not aware of the precise reason for this. Our conversations with processor architects suggest that it may be cache-coherence related; further investigation is needed to confirm this. Nevertheless, our DDR access latency measurement is consistent with other sources [129].

access is 143ns and 135ns, respectively, for CXL-A and CXL-B. We now dive deeper into understanding the root causes of this CXL latency overhead.

The interposer allows us to break down core-to-interposer latency (which we refer to as processor-side latency) and interposer-to-DRAM latency (which we refer to as device-side latency). First, we focus on device-side latency, following which we study processor-side latency.

**Understanding device-side latency.** Since we don't have direct visibility into internals of the CXL device, we infer the device-side latency breakdown using indirect measurements. Our key insight is that different types of requests traverse different subsets of the device's internal datapath. Based on the device architecture shown in Figure 3.2, we can decompose the device-side latency into three key components: (1) CXL port processing: the time taken to deserialize requests and serialize responses; this includes physical layer, link layer and transaction layer processing (2) Interconnect switching: the time taken by deserialized request to traverse the internal interconnect and get enqueued in corresponding MC queue (3) DRAM access: the time taken by the MC to actually execute the request from DRAM. The latency of a memory read request ($L_{rd}$) includes all three of the aforementioned components. The latency of a memory write request ($L_{wr}$), however, includes only components (1) and (2). This is because completions for write requests are issued as soon as the corresponding request is enqueued in the MC queue (§2.3) and therefore do not include DRAM access time. Therefore, $L_{rd} - L_{wr}$ gives us an estimate of the DRAM access latency. Another type of request that can provide further visibility into latency breakdown is CXL link layer retry requests [51]. These requests are used by the link layer for reliability. When a retry request is sent by the host to a device,

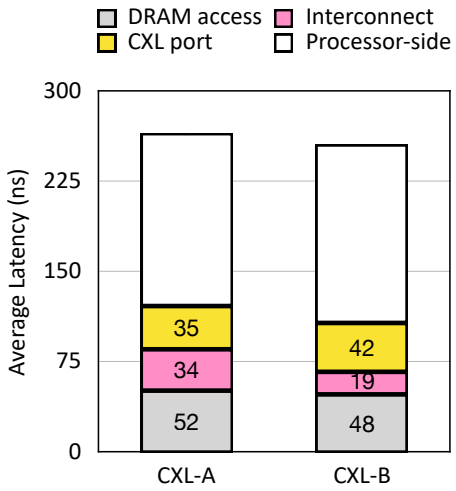


Figure 3.6: CXL latency breakdown.

it is processed entirely within the device's CXL port (traversing only physical and link layers) and a retry ack is sent in response. Hence, the latency of Retry requests (time between retry request being observed by the interposer and retry ack being observed by interposer; call this $L_{retry}$) includes only component (1) and not components (2) and (3). Therefore, $L_{wr} - L_{retry}$ gives us an estimate of component (2) and $L_{retry}$ gives us an estimate of component (1). Interestingly, despite no link errors, we do observe occasional retry request/ack exchanges in our interposer/analyzer traces. This is because, under low load, when the link is idle, it enters a low-power state (Active-State Power Management). Every time this happens, when exiting the low-power state, the link layer is re-initialized and retry request/ack are exchanged as part of the initialization handshake [51]. Overall by measuring $L_{rd}, L_{wr}, L_{retry}$ at the interposer, we can get a breakdown of device-side latency across the three components.

Figure 3.6 shows the device-side latency breakdown obtained using the above methodology. CXL-A has ~8ns lower CXL port latency than CXL-B. This aligns with the fact that CXL-A supports a physical layer optimization called

Sync Header Bypass (SHB) [51, 194] while CXL-B does not[4]. CXL-A's 35ns CXL port latency is still larger than Intel's published estimate of 21 – 25ns [194]. CXL-A interconnect latency component is larger than CXL-B. Our hypothesis is that this is because CXL-A supports two ports while CXL-B supports only one port. CXL-B's interconnect latency is comparable to 15ns estimate used in prior CXL feasibility studies [30, 136]. DRAM access latency for random accesses is expected to be 37 – 53ns for DDR5-4800 and 30 – 44ns for DDR4-3200 [18, 19, 40]. CXL-A and CXL-B DRAM access latency measurements align with these numbers (they are on the upper-end of the expected ranges because average values are impacted by long tails caused by DRAM refreshes as we will discuss in §3.3.2). Overall, the latency overheads introduced by CXL on the device-side (CXL port and interconnect) account for less than half of the end-to-end CXL latency overhead (48% for CXL-A and 45% for CXL-B). The remainder of the CXL latency overhead is incurred on the processor-side.

**Understanding processor-side latency.** Processor-side CXL latency can be decomposed into the following components (1) core to CHA/LLC (2) CHA/LLC to root complex and (3) root complex to interposer. Components (1) and (2) are likely to incur overheads because of the non-localized routing of requests to CHA/LLC slices (as discussed in §3.2). To characterize the impact of non-localized routing on CXL latency, we leverage our knowledge of reverse-engineered mapping of physical addresses to CHA/LLC slices. We add instrumentation to our pointer-chasing benchmark to record physical address and latency on a per-request basis; then by applying the reverse-engineered ad-

[4]Normally, data is transmitted over PCIe physical layer in 130 bit blocks, with each block having a 2-bit sync header followed by 128 bits of payload. Parsing a single CXL flit of 528 bits requires processing sync headers across multiple physical layer blocks which incurs latency. SHB avoids this latency by removing the per-block sync headers and relying on other mechanisms for synchronization.


Figure 3.7: CXL latency broken down by path taken by requests within processor. The CXL device is attached to the IO die that is physically closest to the compute die containing cores 0 – 31 and CHA slices 0 – 39.

dress mapping function, we can compute the average latency per CHA/LLC slice. We run the benchmark on each possible core; this gives us the average latency measurements per (core, CHA/LLC slice). The results are visualized in Figure 3.7. We find that latency variation can be significant across paths—the best-case path latency is 203ns while the worst-case path incurs 289ns. Using this data, we find that non-localized routing of requests (that is, requests being mapped to CHA/LLC slices outside the compute die of the core from which the request originated) results in ~41ns of latency overhead. This accounts for 28.7% and 30.3% of the end-to-end CXL latency overhead for CXL-A and CXL-B respectively. Overall, the device-side latency overheads discussed above and the overhead of non-localized routing on the processor side, taken together, ac-

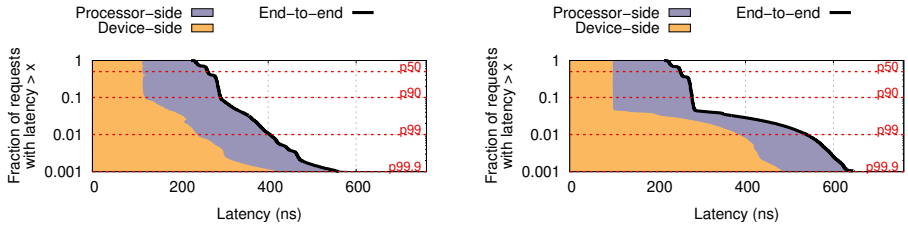



Figure 3.8: Tail latency for CXL memory accesses (a-b; left-right). (a) CXL-A (b) CXL-B. The shaded breakdown is obtained by computing the average of processor-side and device-side latency across 800 requests with latency closest to each percentile.

count for 76.7% and 75.3% of the end-to-end CXL latency overhead for CXL-A and CXL-B respectively. The remaining overhead on the processor-side includes the latency of compute to IO die crossing to reach the root complex and CXL protocol processing time within the root complex itself (we do not have a way to further break down these components).

### 3.3.2 Tail latency

To understand CXL tail latency behavior, we record the latency of individual requests in our pointer chasing workload and analyze the resultant latency distribution. Throughout our analysis, we visualize latency distributions using CCDF plots with log-scale y-axes to highlight tail behavior.

**Device-side is the key contributor to tail latency.** As shown in Figure 3.8, P99 latency is higher than median by 1.54×, 2.14× for CXL-A, CXL-B respectively, and P99.9 latency is 2.14× and 2.57× higher than the median for CXL-A, CXL-B respectively. The breakdowns in the figure clearly show that the tail latency inflation comes from the device-side—relative to the median, the processor-side latency increases by only 17ns and 52ns at the P99 and P99.9 tails for CXL-A,

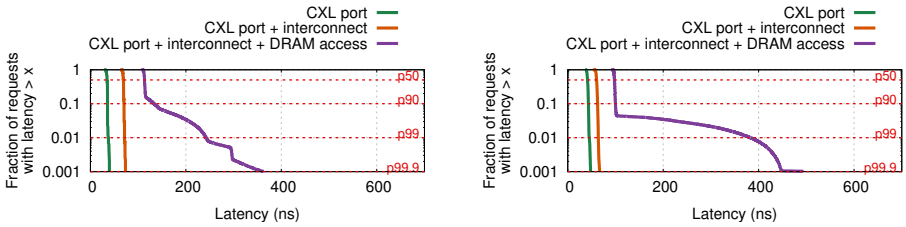



Figure 3.9: CCDF of CXL device-side latency across different components within the device (a-b; left-right). (a) CXL-A (b) CXL-B.

whereas the device-side latency increases by 124ns and 293ns at the same tails (similar observations for CXL-B).

**DRAM access within CXL device is the root-cause of tail latency.** To better understand how different components of the datapath within the device contribute to tail latency inflation, we use the same methodology as in §3.3.1 and independently record the latency distributions of different request types that cover different subsets of the device-internal datapath. Figure 3.9 makes it evident that tail latency inflation is rooted in the DRAM access component—relative to median, the (CXL port + Interconnect + DRAM access) latency distribution incurs 129ns, 249ns inflation at P99, P99.9 for CXL-A and 284ns, 395ns inflation for CXL-B, while the (CXL port + Interconnect) and (CXL port) distributions remain nearly flat. If DRAM access is the cause of tail latency inflation, then one would expect to observe similar tail latency behavior even in the case of local DDR memory accesses. To test this, we compare the latency inflation observed with that of local DDR accesses in servers with the same DDR generation as each of CXL-A and CXL-B separately. The results, shown in Figure 3.10, validate our expectation—comparable latency inflation relative to the median at P99 and P99.9 tails is observed even for local DDR accesses. CXL-A in fact observes smaller P99 latency inflation compared to the corresponding DDR5 servers while observing nearly the same P99.9 latency inflation. For CXL-B, al-

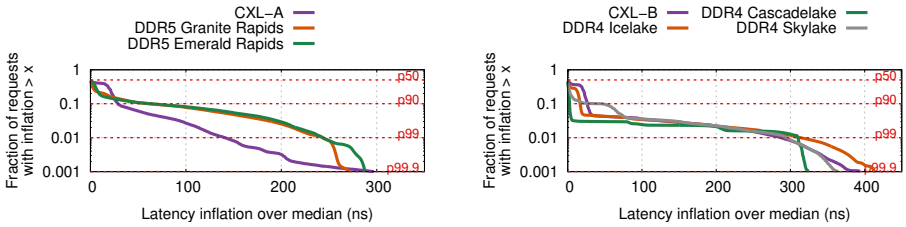



Figure 3.10: CCDF of CXL latency alongside latency of DDR of same generation as in the CXL device across different servers (a-b; left-right). (a) CXL-A (b) CXL-B.

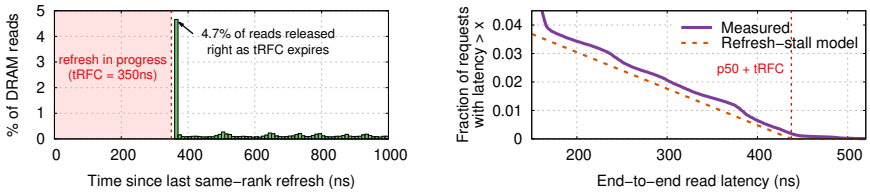



Figure 3.11: DRAM refresh causes tail latency inflation for DDR4-attached memory (a-b; left-right). (a) Distribution of time elapsed since the last same-rank refresh command for every read in the DDR4 analyzer trace. (b) CCDF of observed end-to-end memory access latency, zoomed in to P95 and above, alongside the prediction of a simple model that assumes requests blocked by a refresh arrive uniformly at random within the time duration of the corresponding refresh command.

though there is some variation in the P99.9 latency inflation compared to some DDR4 servers, the general shape of all the latency inflation distributions is remarkably similar. All the above evidence strongly suggests that tail latency inflation is not an artifact introduced by the CXL interconnect or protocol, but rather a property of DRAM access itself.

**Refreshes cause tail latency inflation for DRAM access.** While somewhat orthogonal to our core focus, understanding the root cause of DRAM access tail latency inflation is an interesting exercise. Our conversations with DRAM experts suggested that DRAM refreshes are the likely cause of the tail latency inflation observed in our experiments—DRAM stores each bit as charge on a tiny capacitor, which inevitably leaks over time; to prevent data loss, every cell must be periodically read out and rewritten (i.e., refreshed). As a result, refreshes are a

fundamental aspect of DRAM operation. The memory controller issues refresh commands independently to each rank of a DRAM module at periodic intervals ($t_{REFI}$). While a refresh command is being processed, requests to the corresponding rank are blocked for a certain time duration ($t_{RFC}$). During this period, incoming requests get queued at the memory controller and thus incur higher latency (between 0 and $t_{RFC}$ added latency). To validate the hypothesis that DRAM refreshes are the root cause of tail latency inflation, we used a DRAM analyzer attached to one DDR channel of our DDR4 Skylake server whose latency distribution near perfectly matches that of CXL-B. The analyzer faithfully captures all commands (including both data access and refresh commands) going over the DDR channel without adding any additional latency. Our analysis of the DRAM analyzer traces, shown in Figure 3.11, confirms that refreshes do indeed account for P99 and P99.9 tail latency inflation. One would expect $\frac{t_{RFC}}{t_{REFI}}$ = 4.5% of all requests to get impacted by refreshes. Consistent with this expectation, the trace shows that 4.7% of all requests have time since last refresh within 20ns of $t_{RFC}$ (Figure 3.11(a)). The delay incurred by a request blocked by refresh depends on the time within the $t_{RFC}$ interval during which the request arrives. Fitting a simple model assuming a request arrives uniformly at random within the $t_{RFC}$ interval produces a latency distribution consistent with the observed latency distribution at the tail (P95 and above).

## 3.4 Understanding bandwidth

For our bandwidth experiments, we focus on a single compute die with its 4 DDR memory channels, paired with two CXL-A devices each connected via an x16 link to the same IO die. At first glance this may appear to exercise only a

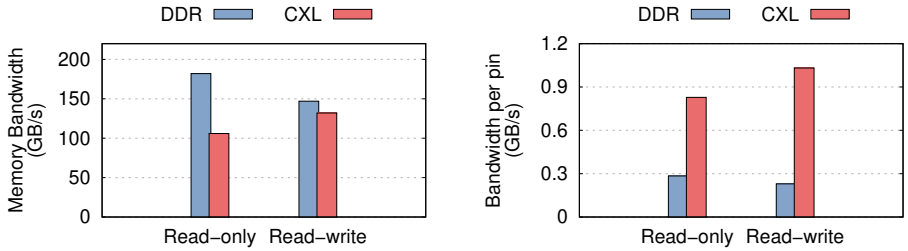



Figure 3.12: CXL versus DDR bandwidth (a-b; left-right). (a) Maximum memory bandwidth achieved over DDR and CXL on our setup (b) Maximum bandwidth-per-pin achieved over DDR and CXL on our setup.

fraction of the socket's maximum DDR bandwidth. However, it exercises the *same fraction* of its maximum CXL bandwidth: the full socket supports up to 6 x16 CXL links (across 2 IO dies) and has 12 DDR channels (across 3 compute dies). Our setup of 4 DDR channels to 2 CXL links preserves the exact same DDR-to-CXL ratio. Because our analysis concerns the *relative* bandwidth of the two interconnects, our smaller-scale setup is representative of a full socket where all DDR channels and all CXL links are utilized. We adopt it because we only had access to two CXL memory devices at the time. We enable CXL interleaving—that is, the processor automatically interleaves memory requests across the two CXL links (akin to how memory requests are interleaved across DDR channels).

We run a workload that sequentially accesses a large buffer placed in DDR/CXL memory with varying number of cores and measure the maximum achieved memory bandwidth in both the DDR and CXL cases. Figure 3.12(a) shows the results with read-only and 1 : 1 read-write access patterns.

**DDR versus CXL for read-only traffic.** For read-only traffic, DDR achieves 87.5% of the theoretical maximum bandwidth (consistent with the typical DDR bandwidth efficiency in contemporary servers). CXL achieves 81.25% of the the-

oretical maximum unidirectional PCIe link bandwidth (128GB/s across the two links). This is very close to the maximum usable bandwidth after taking header overheads into account (83.6%)—read traffic is bottlenecked on response delivery in the device-to-host direction; each 64B response has a 40 bit transaction layer header; in steady-state, the headers for two responses are packed into a single 16B slot (this is referred to as the H5 header packing scheme [57, 194] which we verify is the case from our analyzer traces). This reduces the maximum usable bandwidth for data transfer to 88.8%. Furthermore, to each 64 byte payload, the link layer adds a 2 byte CRC and the physical layer adds a 2 byte protocol ID. This further reduces the maximum usable bandwidth for data transfers to 83.6%.

**CXL can utilize bidirectional PCIe bandwidth for read-write traffic while DDR is limited to half-duplex operation.** For read-write traffic, DDR achieves 70% of the theoretical maximum bandwidth. This is attributed to the fact that DDR can transmit in only one direction at a time, thus requiring the memory controller to switch between executing reads and writes; with each switch incurring bus turnaround overhead during which the channel is idle (discussed in Chapter 2). CXL, on the other hand, is able to utilize bandwidth in both directions of the PCIe links, leading to smaller relative gap between DDR and CXL bandwidth. The bandwidth is, however, smaller than double of what is observed in the read-only case. This is because the DDR channels on the CXL device become bottlenecked. Indeed, the achieved CXL bandwidth is ~73.6% of the theoretical maximum bandwidth offered by the 2x 5600MHz DDR5 channels on each CXL device. We expect that using a CXL device with support for more DDR channels would allow fully saturating bidirectional PCIe bandwidth for read-write workloads.

**CXL is significantly more pin-efficient compared to DDR.** While the absolute bandwidth achieved by CXL is relatively lower compared to DDR, it is important to note that bandwidth per pin in the former is significantly higher. Each DDR5 channel uses 288 pins out of which ~160 go to the processor chip [45]; given 4 channels, DDR consumes a total of 640 processor pins. CXL, on the other hand, in total consumes only 128 pins in our setup (with 4 pins per PCIe lane). Consequently, as shown in Figure 3.12(b), CXL achieves 2.91× higher bandwidth per pin for read-only traffic and 4.49× higher bandwidth per pin for read-write traffic (the reason read-write bandwidth per pin is not exactly double the read-only bandwidth per pin is because of the reason discussed above).

## 3.5 Understanding the impact of data placement on application performance

**Hardware setup.** To understand the impact of data placement across DDR and CXL memory on application performance, we use the same setup as §3.4.

**Workload.** We use the GUPS workload, as in prior work [185, 214, 233]. The workload has a 60GB total working set, comprising a 20GB hot set and a cold set spanning the remaining memory. Each access randomly touches an object in the hot set with 90% probability and in the cold set with 10% probability. For a given DDR memory size, we pack the hot set in DDR and place the remaining data in CXL.

**Default parameters.** Unless otherwise specified, we use a 4KB object size and a 1 : 1 read/write ratio, and run the workload on 30 cores. This leaves 2 cores for

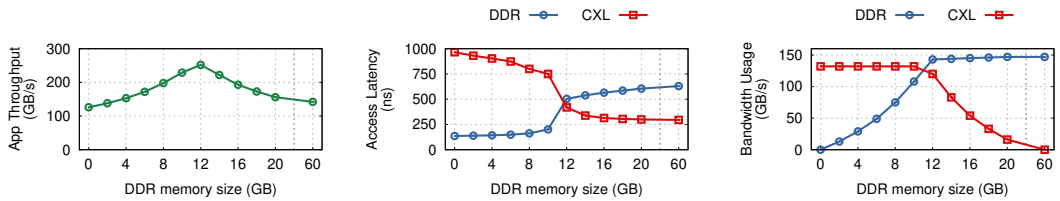



Figure 3.13: Memory disaggregation over cache-coherent interconnect like CXL can improve application performance (a-c; left-right). (a) Steady-state application throughput (b) Access latencies of DDR and CXL memories (c) Bandwidth utilization of DDR and CXL memories.

latency measurement, with one pointer-chasing thread each for DDR and CXL memory.

**Memory disaggregation can improve application performance.** To characterize the impact of memory disaggregation on application performance, we vary the DDR memory size: $x = 0$ corresponds to the extreme where all memory is disaggregated, while $x = 60$ corresponds to the other extreme where no memory is disaggregated. Disaggregating memory over CXL improves application performance by 8-78%, except in the extreme case where all memory is disaggregated, in which case performance degrades by 11%.

Access latency measurements provide insight into these observations. When no memory is disaggregated ($x = 60$), the access latency of DDR memory is 2.14× higher than the CXL access latency. This is because the DDR memory interconnect is contended, causing the access latency of DDR memory to increase by 4.8× compared to the access latency under a single in-flight request. In this regime, offloading memory to CXL reduces the load on the DDR memory interconnect—thereby reducing its access latency—and improves application performance by tapping into the additional bandwidth offered by CXL. As more memory is disaggregated, DDR memory latency continues to reduce

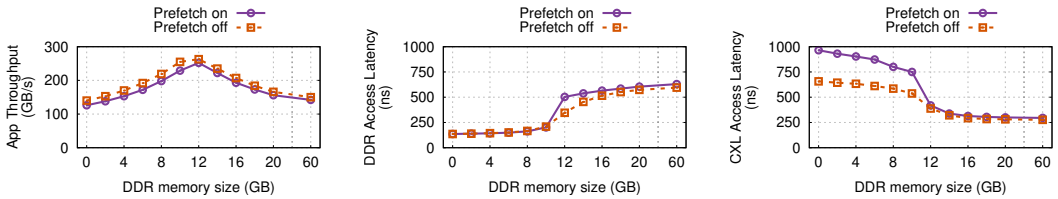



Figure 3.14: Impact of hardware prefetchers (a) Steady-state application throughput (b) Access latencies of DDR memory (c) Access latency of CXL memory.

while CXL latency begins to increase. Beyond a certain point, CXL latency exceeds DDR latency, after which disaggregating more memory reduces application performance.

**Impact of prefetchers.** An interesting observation from the above results is that the access latency in the all-CXL case is 1.53× larger than that observed in the all-DDR case, while the throughput gap is only 1.13×. Our analysis reveals that this is an artifact of hardware prefetchers. Figure 3.14 shows the results of the above experiment with prefetchers turned off. At higher loads, the access latency with prefetchers is significantly higher than without prefetchers for CXL; however, this is not the case for DDR.

Our hypothesis for this phenomenon has to do with dynamic prefetcher throttling (DPT), a processor feature that automatically throttles prefetchers when memory bandwidth is contended. One of the key signals used by DPT is the queue occupancy at the CHA, throttling prefetchers when the occupancy is larger than a threshold. As discussed in §3.2, CXL requests are distributed across a larger number of CHA slices than DDR requests, causing prefetcher throttling to be triggered less frequently in the former case. We observe the prefetcher throttle asserted 90% of the time in the all-DDR case, as opposed to only 17% of the time in the all-CXL case. Prefetchers lead to a larger number of in-flight memory requests per core, which increases load and leads to higher ac-

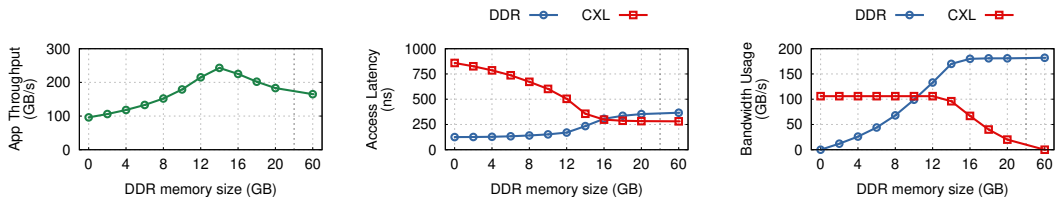



Figure 3.15: Impact of read/write ratio (a-c; left-right). (a) Steady-state application throughput (b) Access latencies of DDR and CXL memories (c) Bandwidth utilization of DDR and CXL memories.

cess latency; we observe 1.48× larger average in-flight requests measured at the CHA in the all-CXL case versus the DDR case with prefetchers on. Despite the higher access latency, given the larger number of in-flight requests, throughput does not degrade proportionally to the latency increase.

While the ineffectiveness of DPT for CXL may not cause a direct performance impact for an individual application, it is likely to have an impact in multitenant scenarios. For example, consider a latency-sensitive application with a random memory access pattern colocated with the above application. The latency-sensitive application would suffer unnecessary throughput degradation because of the high CXL latency, which could have been avoided with proper prefetcher throttling.

**Impact of read/write ratio.** Figure 3.15 shows the results of the above experiment with read-only traffic, as opposed to read-write. We observe two differences compared to the read-write case. First, the maximum throughput improvement achieved is smaller (1.47× as opposed to 1.77× in the read-write case). Second, the maximum amount of data that can be offloaded to CXL before throughput starts degrading is smaller: a DDR memory size of 8GB or smaller results in throughput degradation relative to the all-DDR case. Both of these differences are attributed to the lower available CXL bandwidth, as read-only traf-

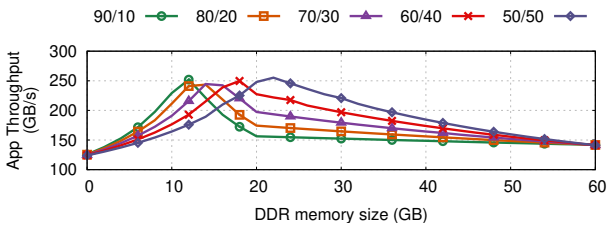



Figure 3.16: Impact of skew in access distribution. Steady-state application throughput as the DDR memory size is varied, for hot/cold access ratios ranging from 90/10 to 50/50.

fic is able to utilize only one direction of the PCIe link (as discussed in §3.4). This reduces the amount of additional bandwidth that application memory accesses can tap into. It also results in the bandwidth getting bottlenecked more easily, which causes CXL latency to increase and exceed DDR latency more quickly.

**Impact of skew in access distribution.** Figure 3.16 shows the results with varying skew in the access distribution while all other parameters are set to their default values. The main difference we observe is that the maximum amount of data that can be offloaded to CXL without performance degradation—and the optimal amount of data to offload for optimal performance—reduces with reducing skew. This is because cold data is first offloaded to CXL before hot data. When skew reduces, the cold data has a relatively higher access frequency; as a result, more load is generated on CXL per unit of cold data offloaded, resulting in CXL latency increasing more quickly.

## 3.6 Related work

Prior work [79,141,208] has characterized latency and bandwidth characteristics of real CXL devices and studied the corresponding impact on application-level

performance. However, they do not provide an in-depth understanding of the hardware-level root causes of CXL latency overheads and/or bandwidth inefficiencies. Our work both corroborates and complements prior observations (by providing deeper understanding) as well as course-corrects and clarifies some prior observations. We now discuss each of the key related prior works.

Demystifying-CXL [208] was the earliest published study of real CXL hardware performance. They report similar average unloaded CXL latency numbers as in our measurements for their best-case ASIC-based devices, but they do not provide a finer-grained breakdown of these latency overheads. They report best-case bandwidth efficiency of 46% for read-only traffic and 65% bandwidth efficiency for read-write traffic; and speculate that the former is due to memory controller inefficiencies. Our analysis in §3.4 shows 81% bandwidth efficiency for read-only traffic and explains the remaining inefficiency via header overheads (ruling out CXL memory controller inefficiency). Our bandwidth efficiency observations for read-write are consistent with theirs. They show that, for bandwidth intensive workloads (DLRM inference), placing some data in CXL can improve performance compared to placing everything in local DDR. Our analysis in §3.5 reinforces these observations while providing deeper insights into the root cause—latency of local DDR exceeding that of CXL. They show CXL inefficiencies related to SNC. When a core evicts an L2 cache line, for DDR it goes to LLC slice within SNC and for CXL it can go to any LLC slice across all SNCs within the socket. They focus on the impact of this phenomenon on LLC isolation across SNCs: applications running in different SNCs can interfere with each other due to LLC capacity contention when memory is placed in CXL. Our analysis in §3.3 complements these observations. We show that the underlying root cause is due to non-localized CHA address mappings, and

characterize the resulting impact on memory access latency (which is complementary to cache capacity isolation impact).

MELODY [141] claims that CXL introduces higher and "unstable" tail latency compared to DDR, and the authors speculate that the cause is associated with CXL flow control, thermal management or CXL memory controller inefficiencies. Our analysis in §3.3 shows that the observed tail latencies (under single in-flight request) are not an artifact of CXL, but rather a property of DRAM access itself, due to refreshes. Extending our tail latency analysis to the case of multiple in-flight requests is an interesting future direction.

Vistara [79] presents measurement of Meta's custom CXL memory controller. They report end-to-end average CXL latency (under single in-flight request) to be 120ns higher than local DDR (130ns for local DDR versus 250ns for CXL). These CXL latency overheads are consistent with our observations on commercially available CXL devices in §3.3. They report idle round trip latency of their custom CXL memory controller to be 50ns. This corresponds to the (CXL port + interconnect) components in our latency breakdowns (§3.3). Our observations are 61 – 69ns across the devices we study, which is comparable but higher than Meta's reported number. They do not provide a finer-grained breakdown of the CXL memory controller latency or an explanation of the remaining CXL latency overhead. Consistent with our observations in §3.3, they report tail latency inflation observed in CXL closely tracks that observed with local DDR. Our analysis explains the root cause of these tails (DRAM refreshes). Their setup reports ~10× lower bandwidth for CXL-attached memory compared to local DDR across different read/write ratios. However, this is an artifact of using low speed DDR4 channels, rather than an inherent limitation of CXL itself.

## 3.7 Implications for the rest of the thesis

Our analysis in §3.3 has implications for the design of disaggregated memory hardware. In particular, reducing the minimum access latency of disaggregated memory requires not just optimizing disaggregated memory controller hardware, but also an end-to-end view of the memory access datapath, including the interplay between cache-coherent interconnects like CXL and the processor interconnect. Our analysis in §3.4 demonstrates that serial interconnects like CXL have the potential to alleviate memory bandwidth bottlenecks imposed by the traditional memory architecture, by providing significantly higher usable bandwidth per processor pin than the DDR memory interconnect. Indeed, our observations in §3.5 show that memory disaggregation over cache-coherent interconnects like CXL can actually improve application performance. Our observations in §3.5 also reveal that, packing the most heavily accessed application data in local memory and offloading the rest to disaggregated memory, does not necessarily maximize application performance. Chapter 4 builds on this observation to re-think OS memory management.

CHAPTER 4

# COLLOID: OS MEMORY MANAGEMENT OVER CACHE-COHERENT INTERCONNECTS

In this chapter, we first demonstrate the impact of our observations from Chapter 3 on existing operating system memory management mechanisms (§4.2). Next, we present the design of Colloid (§4.3), its integration with existing memory management mechanisms (§4.4) and its empirical evaluation (§4.5). Finally, we discuss related work (§4.6) and the implications of this chapter to the rest of this thesis (§4.7).

## 4.1 Overview

Memory disaggregation over cache-coherent interconnects like CXL results in each server or host having multiple memory tiers with different performance characteristics. Local DDR-attached memory forms one tier, and disaggregated memory exposed over cache-coherent interconnects forms a second tier with additional memory bandwidth but with higher minimum access latency (Chapter 3). In addition to memory disaggregation, several other settings also result in each server having multiple memory tiers. Some examples include CXL-based per-server memory capacity expansion without sharing or pooling memory capacity across servers [79,246], and servers that integrate High Bandwidth Memory (HBM) in addition to traditional DRAM [103, 154]. In all of these settings, each server has cache-coherent memory distributed across two or more memory tiers with different performance characteristics. In this chapter, we refer to any such setting as a tiered memory architecture.

Emergence of tiered memory architectures has led to renewed interest in operating system memory management since page placement across memory tiers critically impacts application performance (as shown in Chapter 3). Recent work on memory management for tiered memory architectures [2, 63, 119, 132, 150, 185, 230, 233] has led to many innovative mechanisms for access tracking, page migration, dynamic page size determination, etc.; however, they all use the same page placement algorithm—packing as many hot pages as possible in the memory tier with the lowest unloaded access latency[1] (referred to as the default tier), with the remaining pages placed in alternate tiers. This page placement algorithm is based on a core implicit assumption that, despite serving the hottest pages, memory access latency of the default tier is less than that of alternate tiers. This assumption is far from real: as demonstrated in Chapter 2 and Chapter 3, in the realistic case of multiple in-flight requests, memory access latency can be significantly larger than the unloaded latency, even when the memory interconnect bandwidth is far from saturated. We refer to this regime as memory interconnect contention. Consistent with the observations in Chapter 3, we demonstrate that memory interconnect contention can lead to 5× inflation in access latency of the default tier even under moderate loads. Given the access latency of existing CXL hardware (Chapter 3), such inflation would result in the default tier having 2.5× higher latency than alternate tiers. We show in §4.2 that, under memory interconnect contention, performance of state-of-the-art memory tiering systems can be 2.3× worse than the optimal.

We present Colloid, a tiered memory management mechanism that embodies the principle of balancing access latencies—page placement across tiers

[1]The unloaded access latency is defined as the memory access latency when there is only one request in-flight. In contrast, the loaded access latency is defined as the access latency when there are multiple requests in-flight.

should be performed so as to balance their average (loaded) access latencies. The principle of balancing access latencies builds upon a simple observation: placing more hot pages in the tier with lower access latency may increase the access latency of the tier and decrease the access latency of other tiers (thus, more closely balancing the access latencies of the tiers); however, it will reduce the overall average memory access latency. This observation is important because each processor core is able to keep a bounded number of memory requests in-flight; thus, minimizing average memory access latency directly maximizes memory access throughput and application-level performance (§4.3).

The principle of balancing access latencies provides a unified approach to memory management. First, it naturally captures the *unloaded* latency of the tiers—if the (loaded) access latency of the default tier is smaller than that of alternate tiers, then balancing access latencies requires increasing the fraction of accesses to the default tier, thus converging to the page placement algorithm in existing systems. Second, it captures the fact that memory interconnect contention can happen even when memory bandwidth is far from saturated—memory interconnect contention at any point within the CPU-to-memory datapath will result in access latency increasing for the corresponding tier, and will thus automatically get factored in while making page placement decisions based on the principle.

Memory management based on the principle of balancing access latency changes the core structure of the tiered memory management problem. Indeed, it is no longer optimal to pack as many hot pages as possible in the default tier since placing hot pages in the alternate tier may result in improved application performance. We thus need new mechanisms to measure memory ac-

cess latency, and new page placement algorithms that decide the set of pages to place in each tier. Colloid innovates along both these directions. First, building upon our in-depth understanding of host interconnects from Chapter 2, Colloid demonstrates that modern servers have vantage points within the CPU-to-memory datapath that can be leveraged to perform low-overhead measurements of per-tier queue occupancy and request arrival rates; using Little's Law, this enables Colloid to measure per-tier memory access latency at fine-grained timescales. Second, Colloid presents a new page placement algorithm that takes per-tier memory access latency and per-page access tracking information as input, and uses the principle of balancing access latencies to efficiently decide the set of (hot and cold) pages to place at each tier.

Colloid design is compatible with any cache-coherent tiered memory architecture where different tiers do not share memory channels; this includes local DDR-attached memory, remote socket memory exposed through the processor interconnect, CXL-attached disaggregated memory, and High Bandwidth Memory [103,154]. Colloid design is also compatible with existing mechanisms for access tracking, page migration and page size determination, thus enabling Colloid to easily integrate with existing memory tiering systems. We demonstrate this by integrating Colloid with state-of-the-art memory tiering systems—HeMem [185], TPP [150], and MEMTIS [132]. We evaluate these implementations using real-world applications and using standard workload generators over a range of static and time-varying workloads with varying memory interconnect contention, varying core counts, varying object sizes, varying memory access latencies, and varying read/write characteristics. Across all evaluated scenarios, we find that Colloid enables each system to achieve near-optimal performance, independent of the memory interconnect contention intensity; we

also find that Colloid does not impact the convergence time for the underlying system under time-varying workloads.

Colloid implementation for each system, along with the documentation to reproduce all our results, is available at `https://github.com/host-architecture/colloid`.

## 4.2 Understanding impact of memory interconnect contention on existing memory management systems

In this section, we study three state-of-the-art memory tiering systems—HeMem [185], TPP[2] [150] and MEMTIS [132]—and demonstrate that:

- Under the realistic case of multiple in-flight memory requests, access latency of the default tier can be 5× higher than its unloaded latency. We refer to this regime as the memory interconnect contention regime (defined precisely in §4.3.1). In our setup, the default tier latency in the memory interconnect contention regime can inflate to be 2.4× higher than the alternate tier latency. Based on latencies of real CXL hardware (Chapter 3), a 5× inflation in default tier access latency would correspond to 2.5× higher latency than the alternate tier.
- Existing memory tiering systems pack as many hot pages as possible in the default tier. Under memory interconnect contention, this turns out to be a suboptimal strategy, resulting in these systems performing far from

[2]For TPP, we use Linux kernel v6.3 that includes TPP upstreamed version along with several improvements [95, 125]. We enable Transparent Huge Pages.

optimal—as much as 2.3×, 2.36× and 2.46× worse performance than optimal for HeMem, TPP and MEMTIS, respectively.

**Experimental Setup.** As shown in Chapter 3 and prior work [208], the latency of a CXL-based alternate memory tier varies depending on the memory controller hardware used. To enable studying the impact of different default-to-alternate tier latency ratios, we use remote-socket memory, whose latency we can control (§4.5), as the alternate tier in this chapter. We use a dual-socket server, with each socket having an Intel Xeon Platinum 8362 CPU with 32 cores, 1.25MB L2 cache per-core, 48MB LLC, and 8× 3200MHz DDR4 memory channels with one DIMM attached per channel. Thus, the total theoretical maximum bandwidth across all channels is 205GB/s. Sockets are connected by a UPI link with 75GB/s theoretical maximum bandwidth (in each direction). We use processors only on one of the sockets; the default tier is the locally-attached memory of the socket (32GB capacity, and 70ns unloaded latency), and the alternate tier is the memory attached to the other socket (96GB capacity, and 135ns unloaded latency).

We use the GUPS workload from [185] adapted to our setup. The working set consists of a virtually contiguous buffer of size 72GB. A random 24GB region of this buffer constitutes the hot set; thus, the hot set fits in the default tier but the full working set does not. Cores $1-15$ run one thread each, reading and updating ($1:1$ RW ratio) a 64 byte object chosen at random from the hot set with 90% probability and from the full working set with 10% probability. Cores $31-32$ are used for sampling and migration threads used in memory tiering systems. We do not oversubscribe CPU cores as our evaluation does not focus on evaluating CPU overheads of existing systems; that has already been studied in prior works [132, 185].

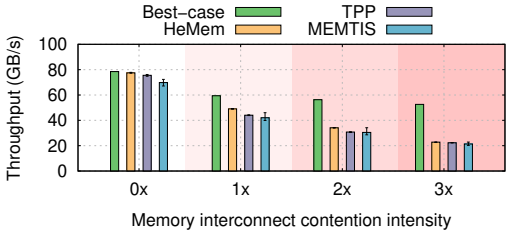



Figure 4.1: Even at moderate memory interconnect contention intensity, existing memory tiering systems achieve performance that is far from optimal. Each bar shows GUPS throughput averaged across 3 runs, with error bars showing the minimum and maximum values across the runs.

We focus on memory interconnect contention on the default tier; memory interconnect contention on the alternate tier does not break the assumption of default tier latency being lower than the alternate tier latency—existing systems already perform ideal page placement under such a regime. To generate controlled memory interconnect contention, as in prior works [3, 4, 43], we use a memory antagonist on cores 16 – 30 that generates sequential $1:1$ read/write memory traffic to a 500MB buffer that is pinned to the default tier memory. To study behavior of the systems in steady-state and provide deeper insights into the observed results, we generate constant memory interconnect contention throughout the experiment; we study the behavior of systems under dynamic changes in memory interconnect contention later (§4.5). Across experiments, we vary the intensity of the memory interconnect contention by varying the number of cores used to run the memory antagonist—$0\times, 1\times, 2\times, 3\times$ intensities correspond to $0, 5, 10, 15$ cores, with memory bandwidth usage of 0%, 51%, 65%, 70%, respectively, when run in isolation. We allow enough time so that each system reaches steady-state, and measure steady-state application throughput.

We determine the best-case memory placement for each configuration by manually placing 0 – 100% of the hot set in the default tier (in increments of 10) using the Linux `mbind` API; the remaining hot set is placed in the alternate tier

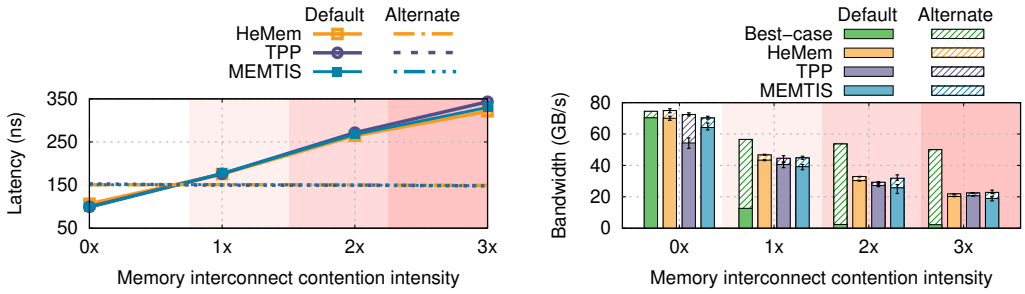



Figure 4.2: Understanding the root cause of suboptimal performance of existing memory tiering systems in Figure 4.1 (a-b; top-bottom). (a) Even with moderate memory interconnect contention intensity, the average memory access latency of the default tier exceeds that of the alternate tier. (b) Existing memory tiering systems place nearly the entire hot set in the default tier independent of memory interconnect contention intensity, while the optimal placement entails placing an increasingly larger fraction of the hot set in the alternate tier with increasing memory interconnect contention.

and any remaining capacity in the default tier is filled with randomly chosen pages from the cold set. We call the highest throughput across these manual placements the best-case application throughput.

Figure 4.1 shows the steady-state throughput for each system alongside the best-case throughput with varying intensity of memory interconnect contention. With 0× memory interconnect contention intensity, HeMem, TPP, and MEMTIS achieve throughput within 1.5%, 4.6% and 10.1% of the best-case respectively. MEMTIS automatically decides whether to use 4KB or 2MB pages for different parts of the working set based on access tracking information. Here, it incurs additional performance degradation because it splits 2MB pages into 4KB pages even though it is not beneficial for this workload. This is because it ends up making hugepage splitting decisions before the workload has reached steady-state and is unable to coalesce pages that have been split; digging deeper, we found that MEMTIS performs page coalescing using an inefficient mecha-

nism of scanning the virtual address space which takes significantly longer than the time it takes for this workload to reach steady-state.

With increasing memory interconnect contention, the throughput achieved by these systems begins to diverge from the best-case. Even with 1× memory interconnect contention intensity, HeMem, TPP and MEMTIS throughput is 1.21×, 1.35×, and 1.41× lower than the best-case, respectively. The throughput gap increases with higher memory interconnect contention intensities, reaching as high as 2.3×, 2.36×, 2.46× for HeMem, TPP and MEMTIS, respectively.

**Under memory interconnect contention, default tier access latency can exceed that of alternate tier.** We measure the average memory access latency for each tier during the above experiments using hardware counters in the processor Caching and Home Agent (CHA); see §4.3.1 for details. Figure 4.2(a) demonstrates that the assumption made in all prior work—despite serving the hottest pages, default tier having lower memory access latency than the alternate tier—does not hold. Indeed, we find that, with memory interconnect contention increasing from 1× to 2× to 3×, the default tier's access latency increases by 2.5×, 3.8× and 5×, respectively. In our setup, this corresponds to default tier access latency exceeding that of the alternate tier by 1.2×, 1.8× and 2.4×, respectively. Access latency increases due to queueing of requests at the memory controller which can happen even when memory bandwidth is far from saturated (see §4.3.1 for more detailed discussion).

**Existing systems continue to greedily place hottest pages in default tier under memory interconnect contention.** Figure 4.2(b) shows the memory bandwidth usage of GUPS across individual tiers for the best-case and for each system, measured using Intel's Memory Bandwidth Monitoring (MBM) feature. For the

best-case, the default tier bandwidth accounts for only 25%, 4.5%, 4% of total bandwidth at 1×, 2× and 3× intensities of memory interconnect contention respectively—corresponding to increasingly larger fractions of the hot set being placed in the alternate tier. Intuitively, when the default tier access latency exceeds that of the alternate tier, it is no longer optimal to place the entire hot set in the default tier. However, for HeMem, TPP and MEMTIS, default tier bandwidth accounts for more than 90%, 75% and 85% of total bandwidth independent of memory interconnect contention intensity—indicating that they always place a majority of the hot set in the default tier. Such memory interconnect contention agnostic page placement is the root cause for their throughput becoming increasingly far from optimal. Specifically, as we discussed in Chapter 2, since each core can maintain a fixed number of in-flight memory requests, per-core throughput ($T$) is given by $\frac{N \cdot 64}{L}$ where $N$ is the number of in-flight requests, $L$ is the access latency and 64 is the size of each memory request in bytes (that is, 1 cacheline). From 0× to 3× memory interconnect contention intensity, default tier access latency increases by ~3.5× and throughput of each of the systems reduces by a similar factor (3.42×, 3.39× and 3.29× for HeMem, TPP and MEMTIS respectively).

## 4.3 Colloid

We now present the Colloid design. We start with the core underlying principle in the Colloid design—the principle of balancing access latencies—and discuss how it provides a unified approach to guide page placement in tiered memory architectures (§4.3.1). We then describe how Colloid realizes this principle into an end-to-end tiered memory management mechanism. Specifically,

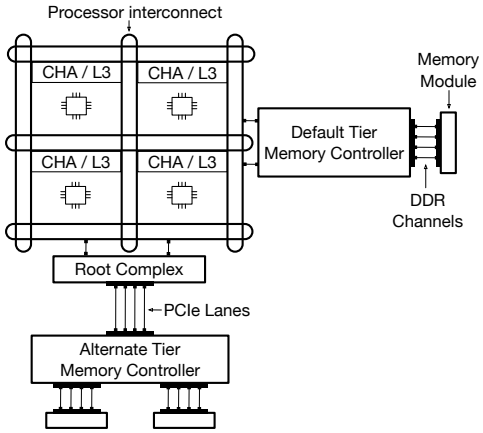



Figure 4.3: Illustration of a tiered memory architecture with two memory tiers. Cores, L3 cache, Caching and Home Agent (CHA), and default tier memory controller are connected through the processor interconnect (we do not make any assumptions on the structure of the processor interconnect; for example, it could span across multiple chiplets as discussed in Chapter 3). Both the L3 cache and CHA are physically distributed into multiple slices (co-located with cores). The physical path between the processor and alternate tier memory controller depends on the type of alternate memory tier. The figure shows CXL-attached memory as an example. Here, the first hop is the root complex on the processor interconnect which is connected to an external memory controller through PCIe lanes.

we describe an efficient mechanism to measure per-tier memory access latency at fine-grained timescales (§4.3.2) and a new algorithm that performs dynamic page placement across tiers based on access latencies (§4.3.3).

We consider a tiered memory architecture illustrated in Figure 4.3. Memory in all tiers is exposed in the host physical address space, and can be accessed by the CPU cores through load/store instructions in a cache-coherent manner with the same memory consistency model. The tier with the smallest unloaded latency is called the default tier, and the others are collectively called alternate tiers. Access to each tier is facilitated through a separate memory controller. All the above properties hold for existing tiered memory architectures where the default tier is local DDR-attached memory, and the alternate tiers are either disaggregated memory exposed over interconnects like CXL, remote socket

memory exposed through the processor interconnect and/or High Bandwidth Memory.

To keep the discussion succinct, we assume that all mechanisms within the memory tiering system (access tracking, latency measurements, page placement, etc.) are performed periodically at fixed time intervals, referred to as quantums. As we discuss in §4.4, Colloid can be easily integrated with systems that use various triggers (*e.g.*, page faults) for access tracking and/or page placement.

### 4.3.1 Access latency is the key

The key conceptual idea underlying the Colloid design is the *principle of balancing access latencies*. The principle suggests that page placement across tiers should be performed so as to balance their average access latencies.

A memory request refers to a memory access issued by a core that misses all levels of the cache hierarchy and is serviced from either the default or the alternate tier. We define the unloaded latency as the latency when there is only one in-flight memory request in the system; and, loaded latency as the latency under multiple concurrent memory requests. As demonstrated in Chapter 2 the loaded latency for both default and alternate tiers can increase beyond their unloaded latency due to contention within the CPU-to-memory datapath:

- Latency can increase due to queueing of requests when the corresponding interconnect bandwidth is saturated [64,80,108,122,123,158,159,161–163,166, 205–208]. Memory bandwidth utilization at the saturation point is hard to

characterize in advance since it can vary by as much as 1.75× depending on whether the workload is read-heavy or write-heavy, and be as much as 2.5× lower than the theoretical maximum bandwidth [208].

- Latency can also increase even when the interconnect bandwidth is far from saturated as observed in Chapter 2. This happens due to contention within the internal hierarchy of memory modules. For example, each DRAM module consists of multiple banks; load imbalance across these banks and lack of locality in access patterns within each bank can result in queueing of requests at the memory controller, leading to latency inflation (detailed discussion in §2.5.1)

We collectively refer to the regime of memory access latency increasing beyond the unloaded latency as memory interconnect contention. We define the access probability of a page during a given quantum as the total number of memory requests to that page normalized by the total number of memory requests across all pages during that quantum. During a given quantum, let $L_D$, $L_A$ be the average access latencies of the default and alternate tiers, respectively, and let $p$ be the sum of access probabilities of pages currently in the default tier.

The principle of balancing access latencies suggests that:

- If $L_D < L_A$, page placement should be adapted to increase $p$ (*e.g.*, by placing more hot pages in the default tier).
- If $L_D > L_A$, page placement should be adapted to reduce $p$ (*e.g.*, by placing more hot pages in the alternate tier).
- If $L_D = L_A$, page placement does not need to be adapted.

The reasoning behind the principle of balancing access latencies is simple. The performance of memory-intensive applications is bound by the overall memory access throughput (rate at which memory requests are serviced). The maximum number of in-flight memory requests that each core can maintain ($N$) is limited by hardware buffer sizes (Line Fill Buffers; as discussed in Chapter 2). As a result, as outlined in §2.4, the average per-core memory access throughput ($T$) is directly related to the average memory access latency ($L$) by $T = \frac{N \cdot 64}{L}$, where 64 is the size of each memory request in bytes (that is, one cacheline). Therefore, minimizing average access latency maximizes throughput. The average memory access latency is given by $p \cdot L_D + (1 - p) \cdot L_A$. If $L_D < L_A$, then increasing $p$ reduces the average access latency; on the other hand, if $L_D > L_A$, then decreasing $p$ reduces the average access latency. Finally, if $L_D = L_A$, then changing $p$ does not change the average access latency. Overall, $L_D = L_A$ is indeed the equilibrium point to minimize average access latency and maximize throughput[3].

The principle of balancing access latencies provides a unified approach to guide page placement across tiers. First, it automatically captures unloaded latencies of each tier—these are included in $L_D$ and $L_A$, respectively. For example, if the gap between unloaded latencies of the tiers is larger, then $L_D$ will remain less than $L_A$ until a larger degree of contention on the default tier, and the principle of balancing access latencies will suggest placing a larger amount of hot pages in the default tier. Second, it captures the maximum theoretical bandwidth of each tier. Higher tier bandwidth usually implies that higher loads can be handled while keeping access latency closer to the unloaded latency (and vice

[3]We don't claim theoretical optimality. Such analysis would require an analytical latency-load model for memory which is hard to build since latency-load characteristics are hardware and workload dependent [64, 161–163]. In §4.5, we empirically demonstrate that the principle of balancing loaded latencies enables memory tiering systems to achieve close to best-case performance for individual workloads.

versa). If the bandwidth of any tier is saturated, then its corresponding access latency will increase, and thus get captured by the principle of balancing access latencies. Third, it captures the impact of memory interconnect contention even when bandwidth is not saturated—queueing at any point in the memory access datapath will result in access latency increase for the corresponding tier, and will thus get factored into page placement decisions based on the principle.

The principle of balancing access latencies naturally generalizes to tiered memory architectures with more than two tiers. If the access latencies of all the tiers are not equal, then the average access latency can be reduced by placing more hot pages in the tier with the smallest access latency. This is because doing so will increase the sum of access probabilities of pages in this tier while correspondingly reducing the sum of access probabilities of pages in other tier(s) which have higher access latencies. Similar reasoning can be applied recursively for the tier with the second smallest access latency and so on. If the access latencies of all the tiers are equal, then adapting page placement will not lead to change in average access latency. Hence, the state of balanced access latency across tiers remains to be the equilibrium point that minimizes average access latency and maximizes throughput.

### 4.3.2 Measuring access latency

Independent of the read/write ratio of the workload, overall memory access throughput primarily depends on the latency of memory read requests. This is because, even for write requests, store instructions first generate memory read upon cache miss to load data into the cache; the data is then updated in the

cache and memory write is processed asynchronously upon cache writeback (§2.3). Thus, the memory access throughput for write requests directly depends on the latency of memory read requests. To that end, we just need mechanisms to measure the latency of memory read requests.

To efficiently measure per-tier access latencies, Colloid builds on two key properties fundamental to any tiered memory architecture. First, the processor must have a routing layer that routes each memory request from the issuing core to the tier its physical address maps to; the routing layer thus observes every request from every core, and knows which tier each request targets. Second, the routing layer must maintain state for every in-flight request (in particular, the identity of the issuing core) in order to route the response back, and this state can be released only when the response returns from the corresponding tier. Consequently, the duration for which a request occupies the routing layer buffers is precisely the time taken to service it from the corresponding tier: queueing at any downstream point in the memory access datapath (*e.g.*, memory controllers, interconnect links) keeps the request allocated in the routing layer for longer, and is thus automatically reflected in routing layer buffer occupancy. Together, these two properties make the routing layer an ideal vantage point for measuring per-tier access latency: it has visibility into requests from all cores to all tiers, and its buffer occupancy captures queueing at any point in the memory access datapath—without requiring visibility into any downstream queue.

Recent generations of Intel and AMD processors provide hardware counters at the routing layer that enable low-overhead measurements of per-tier access latency [14, 100, 101, 104]. For brevity, we focus our discussion on Intel processors, where the routing layer is the Caching and Home Agent (CHA). The

CHA abstracts away memory tiers from the rest of the system while handling cache-coherence. As shown in Figure 4.3, the CHA is physically partitioned into multiple slices (colocated with cores), each of which owns a disjoint subset of the host physical address space. Upon an L1/L2 cache miss, memory requests are first forwarded to their corresponding CHA slice based on their physical address. The CHA looks up the L3 cache. Upon a miss, the CHA queues the request in its buffers and then forwards the request to the default or alternate tier based on the physical address. The request remains queued at the CHA until it has been serviced from the corresponding tier.

CHA hardware counters enable low-overhead measurements of queue occupancy and request arrival rates on a per-request type (read/write) and per-tier basis for local DDR-attached memory, remote socket memory exposed through the processor interconnect, CXL-attached memory, as well as High Bandwidth Memory. These measurements can be performed at much finer-grained timescales (as low as 1 microsecond) than the timescales at which memory tiering systems typically make page placement decisions. During a given time quantum, let $O_D$, $O_A$ be the average queue occupancy of default and alternate tier requests, respectively, and let $R_D$, $R_A$ be the average arrival rate of requests to the default and alternate tiers (number of memory requests that arrived during the quantum divided by the quantum duration). $R_D$ and $R_A$ are directly related to $p$ (sum of access probabilities of pages in the default tier): $p = \frac{R_D}{R_D+R_A}$. Colloid uses Little's Law to measure the access latency ($L_D$, $L_A$) of each tier: $L_D = O_D/R_D$, $L_A = O_A/R_A$. Since Little's Law applies to any system where queues do not grow unboundedly (without any assumptions on arrival, service behaviors, scheduling policies etc.), applying it at the CHA gives us the average CHA to memory latency; §2.6 provides in-depth validation of Little's

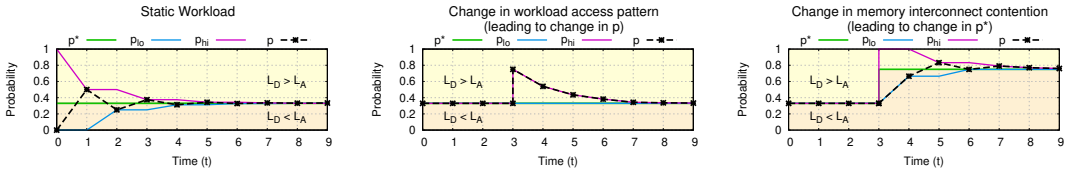



Figure 4.4: Conceptual illustration of Colloid page placement algorithm (a-c; left-right). Each figure shows the change in sum of access probabilities of pages in default tier $p$ over time, along with the equilibrium point $p^*$, low watermark $p_{lo}$, and high watermark $p_{hi}$. Colloid page placement algorithm adapts $p$, $p_{lo}$ and $p_{hi}$ so as to maintain two invariants: both $p$ and $p^*$ are contained between $p_{lo}$ and $p_{hi}$ at all times, and the gap between $p_{lo}$ and $p_{hi}$ reduces over time. (a) Static workload: $p$ eventually converges to $p^*$ (b) Upon sudden change in $p$ at $t = 3$, $p_{hi}$ is *updated*, after which $p$ eventually converges to $p^*$ (c) Upon sudden change in $p^*$ at $t = 3$, $p_{hi}$ is *reset*, after which $p$ eventually converges to $p^*$.

law based memory access latency measurements. The only missing component in measured $L_D$, $L_A$ is CPU to CHA latency. This accounts for only a tiny fraction of the CPU to memory access latency and can thus be ignored. For example, on our hardware setup, out of ~70ns unloaded latency for default tier, ~5ns is CPU to CHA and ~65ns is CHA to DRAM; under memory interconnect contention, the latter will become even higher, making the former an even smaller fraction of the overall latency. We apply Exponentially Weighted Moving Averaging (EWMA) on both the occupancy and rate measurements to smooth noise in the signals. This trades-off slightly higher reaction time during workload changes for better stability.

### 4.3.3 Page placement algorithm

Colloid page placement algorithm takes per-tier access latency and per-page access tracking information as input, and adapts page placement across tiers.

**Overview of the Colloid page placement algorithm.** Algorithm 1 shows the end-to-end Colloid page placement algorithm. At the beginning of each quan-

tum, it obtains the per-tier average queue occupancy ($O_D$, $O_A$) and average request rates ($R_D$, $R_A$) over the duration of the previous time quantum, computes the per-tier access latencies ($L_D$, $L_A$), and current value of the sum of access probabilities of pages in the default tier $p$. It then decides where to migrate pages based on the principle of balancing access latencies (§4.3.1)—if default tier access latency is smaller than that of the alternate tier, then it promotes hot pages from the alternate tier to the default tier; otherwise it demotes hot pages from the default tier to the alternate tier (Lines 5-8). Next, it determines what pages to migrate in three steps. First, it computes how much access probability to shift between the tiers, denoted as $\Delta p$ (Line 9). Second, it computes a migration limit. Third, it executes a page finding procedure (Line 10), whose goal is to find a set of pages in default/alternate tier for demotion/promotion under the following two constraints: (1) their sum of access probabilities is less than or equal to $\Delta p$; and (2) their sum of sizes in bytes does not exceed the migration limit or the available capacity in the destination tier. Depending on the underlying system and access tracking mechanism, one can implement different page finding procedures, which we describe in §4.4. Finally, it migrates these pages through an underlying page migration mechanism.

**Desired shift in per-tier access probability.** In existing memory tiering systems, the decision on the set of hot pages to migrate during each quantum is trivial—simply migrate as many hot pages as possible (while respecting migration rate limits). For Colloid, however, the principle of balancing access latencies changes the structure of the problem—the equilibrium point may correspond to placing a subset of hot pages in the default tier and a subset of the hot pages in the alternate tier. As a result, the decision on the set of hot pages to migrate during each quantum is non-trivial. Migrating too many hot pages may result

**Algorithm 1 : Colloid page placement algorithm.**

[Input parameter] $M$: Migration limit per-quantum (in bytes)
Every quantum:

1: $O_D$, $R_D$: Measured default tier queue occupancy, rate
2: $O_A$, $R_A$: Measured alternate tier queue occupancy, rate
3: $L_D \leftarrow \frac{O_D}{R_D}$ and $L_A \leftarrow \frac{O_A}{R_A}$
4: $p \leftarrow \frac{R_D}{R_D+R_A}$
5: **if** $L_D < L_A$ **then**
6: mode $\leftarrow$ `promotion`
7: **else**
8: mode $\leftarrow$ `demotion`
9: $\Delta p \leftarrow$ COMPUTESHIFT($p$, $L_D$, $L_A$)
10: $S \leftarrow$ FINDPAGES(mode, $\Delta p$, $min\,(\Delta p(R_D + R_A), M)$)
11: **if** mode is `promotion` **then**
12: promote pages in $S$ from alternate to default
13: **else**
14: demote pages in $S$ from default to alternate

in oscillations around the equilibrium point. Migrating too few hot pages may result in longer time to converge to the equilibrium point. Colloid determines the set of hot pages to migrate by computing the desired shift in per-tier access probabilities.

**Algorithm 2 : Computing desired shift in access probability.**

/* Initialize $p_{lo} \leftarrow 0$ and $p_{hi} \leftarrow 1$ */

1: **procedure** COMPUTESHIFT($p$, $L_D$, $L_A$)
2: **if** $|L_D - L_A| < \delta \cdot L_D$ **then return** 0
3: **else**
4: **if** $L_D < L_A$ **then** $p_{lo} \leftarrow p$ **else** $p_{hi} \leftarrow p$
5: **if** $p_{hi} < p_{lo} + \epsilon$ **then**
6: **if** $L_D < L_A$ **then** $p_{hi} \leftarrow 1$ **else** $p_{lo} \leftarrow 0$
**return** $\left|\frac{p_{lo}+p_{hi}}{2} - p\right|$

To find the desired shift in access probability, Colloid executes a binary-search style procedure (Algorithm 2) using two watermarks, $p_{lo}$ and $p_{hi}$. In-

tuitively, $p_{hi}$ upper bounds the sum of access probabilities (known so far based on measurements during previous quanta) for which the default tier access latency *may be* smaller than that of alternate tier; $p_{lo}$, on the other hand, bounds the sum of access probabilities (known so far based on measurements during previous quanta) for which the default tier access latency is *definitely* smaller than that of alternate tier. The above intuitive definitions suggest that $p_{lo}$ should be initialized to 0 and $p_{hi}$ should be initialized to 1, based on the unloaded access latencies of the tiers. Moreover, if there is a feasible equilibrium point $p^*$, it will always be somewhere between the low and high watermarks—that is, $p_{lo} \leq p^* \leq p_{hi}$. During each quantum, Colloid tries to reduce the gap between $p_{lo}$ and $p_{hi}$, thus moving the system closer to the equilibrium point. If default tier access latency is smaller than that of alternate tier, then $p_{lo}$ is updated to $p$, otherwise, $p_{hi}$ is updated to $p$. Colloid then moves the system towards the midpoint of $p_{lo}$ and $p_{hi}$. To that end, the desired shift in access probability across the tiers is computed as $\left|\frac{p_{lo}+p_{hi}}{2} - p\right|$.

As illustrated in Figure 4.4(a), for any given static workload, the above procedure will converge to the equilibrium point if one exists. This is because the gap between $p_{lo}$ and $p_{hi}$ reduces during each quantum, while the invariants $p_{lo} \leq p \leq p_{hi}$ and $p_{lo} \leq p^* \leq p_{hi}$ continue to hold. As a result, $p$ eventually converges to $p^*$. If the access latency of the default tier is lower than the alternate tier even when $p = 1$, then Colloid converges to the optimal operating point of $p = 1$.

The key challenge is to handle dynamic time-varying workloads. Such workloads can result in two unexpected changes: (1) changes in workload access patterns can result in abrupt change in $p$ value violating the $p_{lo} \leq p \leq p_{hi}$ in-

variant; and, (2) changes in memory interconnect contention can result in abrupt change in $p^*$ value violating the $p_{lo} \leq p^* \leq p_{hi}$ invariant. As illustrated in Figure 4.4(b), changes in $p$ are automatically handled by the above procedure, since the watermarks are updated before computing the $\Delta p$ value.

Changes in $p^*$, however, require more care. For example, say $p^*$ abruptly becomes larger than $p_{hi}$ due to change in memory interconnect contention, as illustrated at $t = 3$ in Figure 4.4(c). Since the above procedure ensures that $p$ remains lower than $p_{hi}$ at all times and since $p^*$ is now larger than $p_{hi}$, $p$ will fail to converge to the new $p^*$. To handle dynamic changes in the equilibrium point, Colloid checks if the watermarks have gotten very close to each other and if the system has not yet reached the equilibrium point (that is, the access latencies of the tiers are not balanced); if so, Colloid resets the watermarks based on the access latencies of the tiers. Specifically, if $p_{lo}$ is very close to $p_{hi}$ and the default tier access latency is smaller than that of the alternate tier, then $p^*$ has likely exceeded $p_{hi}$; thus Colloid resets $p_{hi}$ to 1. Similarly, if $p_{lo}$ is very close to $p_{hi}$ and default tier access latency is larger than that of alternate tier, then $p^*$ has likely become smaller than $p_{lo}$; thus Colloid resets $p_{lo}$ to 0. Resetting the watermarks when needed enables Colloid to converge to the equilibrium point even when the equilibrium point shifts due to changes in memory interconnect contention. This is illustrated in Figure 4.4(c). To quantify the gap between the watermarks, we use a threshold $\epsilon$ ($0 < \epsilon < 1$), and to determine whether the system is close to the equilibrium point we check whether the access latencies of the tiers are within a factor $\delta$ of each other ($0 < \delta < 1$). Given fixed $\delta$, increasing $\epsilon$ leads to faster detection of dynamic workload changes at the cost of worse stability. Given fixed $\epsilon$, increasing $\delta$ leads to better stability at the cost of suboptimal steady-state throughput.

**Dynamic migration limit.** As Colloid converges closer to the equilibrium point, it must carefully navigate the trade-off between the desired shift in access probability $\Delta p$ and the impact of additional memory traffic generated by page migrations on application performance. During a given quantum, if $\Delta p$ is small and there are many pages each with tiny access probability (*e.g.*, if $\Delta p = 0.01$, and there are 1000 pages with probability 0.00001 each), Colloid may end up unnecessarily migrating a large number of pages. This may lead to a large volume of migration traffic, causing oscillations around the equilibrium point, and hurting application performance. To address this, Colloid introduces a dynamic migration limit proportional to $\Delta p$. If migration traffic is larger than the desired perturbation in rates $\Delta p(R_D + R_A)$, then it is not ideal to execute such a migration. Therefore, Colloid sets the migration limit to the minimum of $\Delta p(R_D + R_A)$ and a static limit based on maximum rate-limit for migration traffic, which is a configurable parameter in existing systems.

**Selecting pages to migrate.** Given a desired shift in access probability $\Delta p$, Colloid uses the access tracking mechanism in the underlying system to find a small set of pages that meet the bound on the desired shift in access probability (§4.4).

## 4.4 Integrating Colloid with existing memory management systems

Colloid leverages existing access tracking and page migration mechanisms, thus enabling easy integration with existing memory tiering systems. We integrate Colloid with three state-of-the-art memory tiering systems, HeMem [185],

MEMTIS [132] and TPP [150]. This section provides the most interesting system-specific implementation details.

Integrating Colloid with any system requires implementing access latency measurement (§4.3.1), Colloid page placement algorithm (§4.3.3), and a page finding procedure based on available access tracking information. Colloid implementation uses the same trigger for page placement decisions (*e.g.*, periodically, or on page faults), and the same page migration limits as the underlying system.

**HeMem with Colloid.** HeMem uses Processor Event-Based Sampling (PEBS) to compute per-page access frequencies. In particular, it uses a busy polling thread that reads PEBS access samples with a fixed sampling rate, and updates per-page frequency counts. It maintains a hot list and cold list of pages for each tier. Pages are placed in the hot list if their access frequency count exceeds a fixed threshold. Pages are cooled by halving their frequency count whenever the frequency count of any page reaches a certain threshold (`COOLING_THRESHOLD`). Page migrations are performed asynchronously by a migration thread using a quantum of 10ms.

We implement Colloid on top of HeMem with 520 lines of code. Colloid access latency measurements are performed on HeMem's migration thread—CHA hardware counters are sampled every quantum to obtain average queue occupancy and average rate of requests for each tier. Colloid page placement algorithm, that replaces the page placement algorithm of HeMem, is also implemented on the migration thread. We leverage HeMem's per-page frequency counts to compute the access probability of each page. Specifically, the access probability of each page is obtained by dividing its frequency count by the cu-

mulative frequency count across all pages. To efficiently identify pages that correspond to $\Delta p$ access probability, we extend HeMem's hot/cold lists: rather than binary hot/cold lists, we split the frequency space (0-`COOLING_THRESHOLD`) into equal sized bins and maintain a separate page list per bin. We use 5 bins by default. These lists are updated in the exact same fashion as HeMem's original hot/cold lists. Using these, we can easily determine which pages to migrate: we iterate over bins to find pages whose sum of access probability is less than or equal to $\Delta p$, until either $\Delta p$ is satisfied, migration limit is hit, or there are no feasible page choices.

**MEMTIS with Colloid.** MEMTIS is similar to HeMem, with four key differences. First, it uses a dynamic sampling rate for PEBS to reduce CPU overhead. Second, it uses a dynamic threshold (based on the measured access distribution) to determine the hot and the cold page list for each tier. Third, it performs promotion and demotion using separate per-tier `kmigraterd` threads using a quantum of 500ms. Finally, MEMTIS performs page size determination—based on certain heuristics, MEMTIS coalesces pages into hugepages using a background thread, and splits hugepages into pages using `kmigraterd` threads.

We implement Colloid on top of MEMTIS with 411 lines of code. Since Colloid design is agnostic to the sampling rate used to maintain per-page frequency counts, our implementation on top of MEMTIS computes per-page access probabilities similar to our implementation on top of HeMem. We do not modify MEMTIS hot/cold list management, and simply use the per-tier hot lists to select pages for migration. We implement Colloid latency measurement and page placement algorithm on the alternate tier `kmigraterd` thread (default tier `kmigraterd` is unmodified and demotes cold pages based on capacity con-

straints as before). To determine which pages to migrate, we scan the corresponding tier's hot list and pick pages until either $\Delta p$ is satisfied or the migration limit is hit. To handle different page sizes, we simply need to take each individual page's size into account when checking the migration limit.

**TPP with Colloid.** TPP periodically scans process page tables and marks pages with a special protection bit. Subsequent accesses to these pages result in a hint page fault. Hot and cold pages are determined using time-to-fault, that is, time duration between a page being marked and subsequent hint fault being triggered. A page is deemed to be hot if the time-to-fault is larger than a dynamically adapted threshold, and cold otherwise. Upon hint fault for a page in the alternate tier, it is synchronously promoted to the default tier if the page is deemed to be hot. Demotion of cold pages from default tier to alternate tier happens asynchronously through the `kswapd` thread which demotes pages based on capacity watermarks. Cold pages for demotion are picked from the kernel's inactive list (which is maintained based on page access bits).

We implement Colloid on top of TPP in Linux kernel v6.3 with ~315 lines of code. We implement Colloid access latency measurement in a kernel module that runs a spin polling thread which samples CHA counters at microsecond-scale, and exposes occupancy and rate metrics to the core kernel. This requires dedicating one core similar to vanilla HeMem. To compute per-page access probability, we use time-to-fault as a proxy—pages with higher access probability are likely to hint fault more quickly than those with lower access probability. Specifically, in a stream of requests, the average number of requests before a page with access probability $p$ is accessed is $\frac{1}{p}$. If $r$ is the current rate of requests to the corresponding tier, we get an average inter-request time of $\frac{1}{r}$. Thus, the

expected time before a page is accessed (or equivalently, the time-to-fault for a page) is given by $\Delta t = \frac{1}{p \cdot r}$. Rearranging the equation, the access probability of a page can be calculated as $p = \frac{1}{\Delta t \cdot r}$.

Colloid page placement algorithm is implemented by modifying the hint fault handler. Upon hint fault of page in alternate tier, the page is promoted to default tier if current access latency of alternate tier is larger than default tier and the page's access probability is less than or equal to $\Delta p$ (Algorithm 1). To enable demotion of hot pages from the default tier to the alternate tier, we enable hint faults on the default tier pages. Upon hint fault of default tier page, the page is demoted to alternate tier if default tier access latency is larger than alternate tier access latency, and its access probability is less than or equal to $\Delta p$. Demotion of cold pages from default tier to alternate tier via `kswapd` continues as before to meet capacity constraints.

## 4.5 Evaluation

We now evaluate the performance of three state-of-the-art memory tiering systems—HeMem, MEMTIS and TPP—with and without Colloid. Unless specified otherwise, all parameters of all systems are set to their default values. For TPP, by default Transparent Huge Pages (THP) are enabled; we also present results with THP disabled (referred to as TPP w/o THP). For Colloid, we set $\epsilon = 0.01$ and $\delta = 0.05$ by default. Throughout, we use the same setup as in §4.2.

We focus on two key metrics: steady-state application throughput, and convergence time upon changes in workload and memory interconnect contention intensity. For real applications, we measure application-specific performance

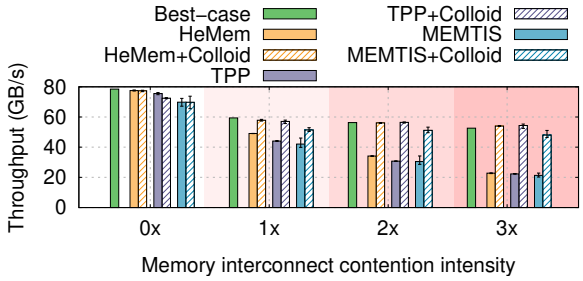



Figure 4.5: Colloid enables each system to achieve near-optimal performance, independent of the memory interconnect contention intensity.

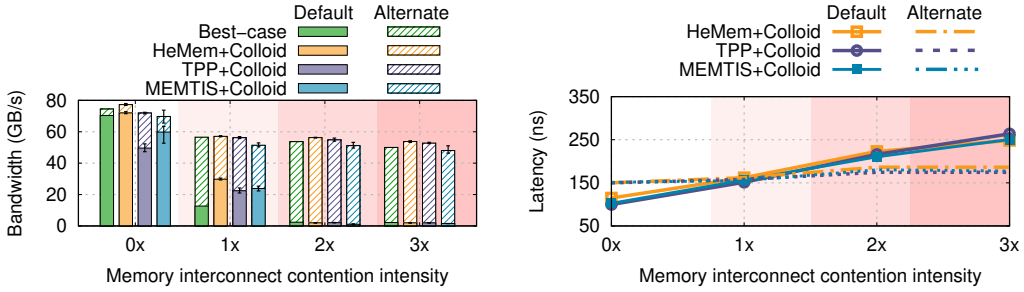



Figure 4.6: Understanding Colloid benefits (a-b; top-bottom). (a) Colloid enables each system to better balance hot pages across tiers based on the memory interconnect contention, similar to the best-case placement; (b) Colloid reduces the gap between the access latencies across tiers compared to Figure 4.2(a).

metrics that are described inline.

**Colloid impact on steady-state throughput.** Figure 4.5 shows the steady-state throughput achieved by each system with and without Colloid, along with the best-case throughput for the same GUPS workload as in Figure 4.1. With 0× memory interconnect contention intensity, performance with Colloid matches performance without Colloid for all systems. With increasing memory interconnect contention, Colloid provides increasingly larger benefits in terms of steady-state throughput: 1.2 – 2.3× for HeMem, 1.35 – 2.35× for TPP and 1.29 – 2.3× for MEMTIS. Independent of the intensity of memory interconnect contention, Colloid enables HeMem, TPP and MEMTIS to achieve close to the best-case performance (within 3%, 8% and 13%, respectively).

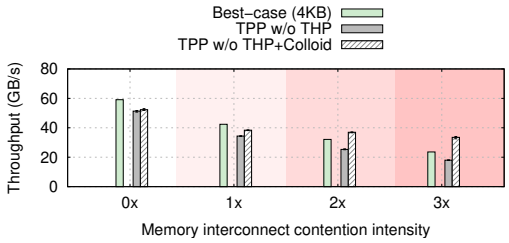



Figure 4.7: Throughput achieved by TPP without THP, with and without Colloid, alongside the best-case manual placement with 4KB pages. We discuss the reason for TPP w/o THP+Colloid throughput being larger than best-case in some cases in the text.

**Understanding Colloid benefits.** Figure 4.6(a) shows that Colloid enables each system to better adapt page placement based on memory interconnect contention; intuitively, this is because Colloid page placement algorithm is able to determine the precise set of hot pages to place in each tier so as to balance access latencies as shown in Figure 4.6(b).

With 0× memory interconnect contention intensity, Colloid behavior is similar to the underlying system (Figure 4.2(b))—it places nearly the entire hot set in the default tier since the default tier access latency remains lower than that of the alternate tier. For $1\times, 2\times$ and $3\times$ memory interconnect contention intensity, unlike the results in Figure 4.2(b), each system now places an increasingly larger fraction of the hot set in the alternate tier, similar to the best-case placement. With 1× memory interconnect contention intensity, it balances the hot set across the tiers in order to equalize their access latencies. With 2×, and 3× memory interconnect contention intensity, Colloid places the entire hot set in the alternate tier. In these cases, even after doing so, the default tier access latency remains higher than that of the alternate tier, but the gap between the two is significantly lower compared to the Figure 4.2(a) measurements without Colloid for all systems.

**Impact of page size.** Figure 4.7 shows the steady-state throughput achieved by TPP, with and without Colloid, with THP disabled, resulting in 4KB pages instead of 2MB pages. For consistency, we compare these numbers with the performance of the best-case manual placement with 4KB pages.

At 0× memory interconnect contention intensity, best-case throughput with 4KB pages is 60GB/s which is significantly lower than the best-case throughput of 79GB/s for 2MB pages. This is because of TLB misses which add to memory access latency (last-level TLB miss ratio is 76% with 4KB pages versus 8% with 2MB pages). TPP w/o THP throughput is 14% lower than corresponding best-case at 0× memory interconnect contention intensity (due to page migration overheads associated with smaller size pages). With increasing memory interconnect contention intensity, the gap from best-case increases reaching up to 1.31× at 3× intensity. TPP w/o THP throughput is similar to HeMem, TPP and MEMTIS for 3× memory interconnect contention intensity, because impact of larger default tier memory access latencies masks page migration overheads.

TPP w/o THP+Colloid throughput is within 10% of best-case with 1× memory interconnect contention intensity and higher than the corresponding best-case with 2× and 3× memory interconnect contention intensity; the latter observation is because our methodology for estimating the best-case does not take page table page placement across tiers into account. Using 4KB pages results in a large number of TLB misses, resulting in memory traffic for page-table accesses[4]. For best-case, while the data pages in the hot set are bound to the alternate tier, their corresponding page table pages get allocated in the default tier (this is an implementation artifact; we access all the hot pages first to en-

[4]With 4KB pages, under 0× memory interconnect contention intensity, even when entire working set is bound to alternate tier, ~ 17% of memory traffic goes to the default tier.

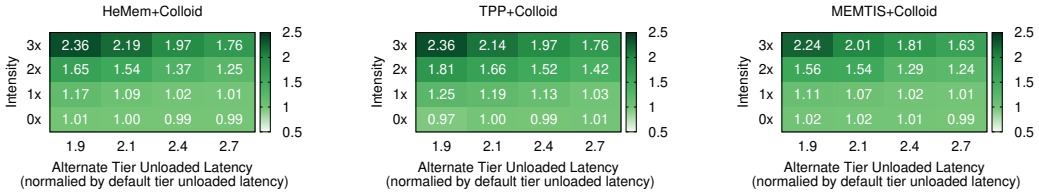



Figure 4.8: Colloid enables performance benefits for the underlying system even with larger alternate tier unloaded latency (a-c; left-right). Each cell in the heatmap shows performance improvement enabled by Colloid (throughput of the underlying system with Colloid normalized by throughput without Colloid) for a specific alternate tier unloaded latency and memory interconnect contention intensity.

sure that they are placed in accordance with their `mbind` policy). Memory contention impacts these page table accesses, thus reducing throughput. In TPP w/o THP+Colloid, the page table pages of the hot set get placed in the alternate tier. Under memory contention, when the system places all the hot data pages in the alternate tier, both data page and page table accesses go to the alternate tier, leading to higher performance.

We now perform sensitivity analysis for Colloid with varying alternate tier unloaded access latency, varying object sizes, varying number of application cores, and varying read/write ratios. Each of these parameters impacts the optimal operating point in terms of the set of hot pages that must be placed in each tier to achieve optimal performance, thus enabling us to understand Colloid effectiveness across the spectrum.

**Impact of alternate tier unloaded latency.** Existing CXL-attached memory has 2× higher unloaded latency relative to the default tier for ASIC-based memory controllers [208, 246]. In our evaluation so far, the unloaded latency of our alternate tier is 1.9× the default tier latency, roughly matching the current CXL-attached memory latency ratio. To better understand Colloid behavior under a wider range of realistic unloaded latencies of tiered memory deployments,

we increase the unloaded latency on our servers by reducing the uncore frequency of the remote socket *only for this experiment*. This enables us to vary the unloaded latency of the alternate tier from 1.9 – 2.7× of the default tier latency. Reducing uncore frequency has the side effect of reducing alternate tier bandwidth in addition to increasing unloaded latency. Such bandwidth reduction only hurts Colloid (that is, the reported gains are conservative)—for real hardware with comparable unloaded latency, higher bandwidth would allow Colloid to achieve better results: it will be able to place larger number of hot pages in the alternate tier since loaded latency will not increase as quickly due to higher bandwidth of the alternate tier.

Figure 4.8 demonstrates that Colloid enables significant performance benefits for existing memory tiering systems even when the alternate tier unloaded latency is as high as 2.7× that of the default tier. As expected, Colloid enables larger benefits for higher memory interconnect contention intensities. For a fixed memory interconnect contention intensity, the benefits of Colloid reduce with increasing alternate tier unloaded latency because the throughput benefits achieved from placing hot pages in the alternate tier reduce with higher access latency; in addition, a relatively smaller number of hot pages can be placed in the alternate tier before its access latency exceeds that of the default tier. Nevertheless, even at the maximum alternate tier unloaded latency (2.7× of default tier), Colloid still achieves 1.01 – 1.76×, 1.03 – 1.76× and 1.01 – 1.63× performance improvement for HeMem, TPP and MEMTIS, respectively.

**Impact of object size.** Figure 4.9 shows the benefits of Colloid with the GUPS object size varying from 64 to 4096 bytes, with all other parameters fixed to their default values. For object sizes 256 bytes and above, Colloid provides per-

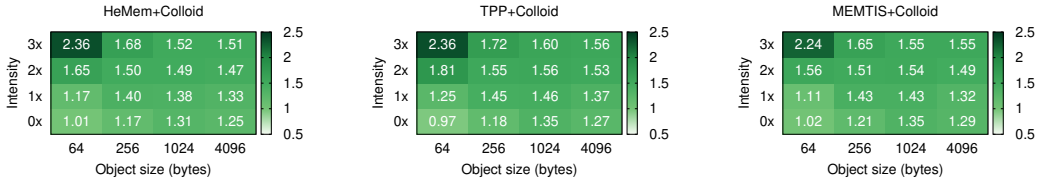



Figure 4.9: Colloid enables performance benefits for the underlying system, independent of the object size. With larger object sizes, Colloid enables performance improvements even at low memory interconnect contention intensity. (a-c; left-right). Each cell in the heatmap shows performance improvement enabled by Colloid (throughput of the underlying system with Colloid normalized by throughput without Colloid) for a specific object size and memory interconnect contention intensity.

formance improvements even for 0× memory interconnect contention intensity (1.17 – 1.31× for HeMem, 1.18 – 1.35× for TPP and 1.21 – 1.35× for MEMTIS). This is because larger object sizes make the workload access pattern more sequential; as a result, hardware prefetchers perform better and the effective per-core parallelism for memory objects is increased. For example, with HeMem at 0× intensity, the average number of in-flight L3 misses per core at the CHA—which corresponds to effective per-core parallelism—is 2.82× higher for 4096 byte object size compared to 64 byte object size. Thus, the application becomes more memory intensive, causing the default tier access latency to exceed that of the alternate tier even at 0× intensity. For example, with HeMem at 0× intensity, the default tier access latency exceeds the alternate tier latency by 1.77× for 4096 byte object size enabling Colloid to provide performance improvement even at 0× intensity.

With higher memory interconnect contention intensity, Colloid benefits reduce a little bit with increase in object sizes. This is because the alternate tier memory interconnect becomes contended, thus limiting the additional throughput that Colloid can achieve by placing hot pages in the alternate tier. For example, with HeMem+Colloid at 3× intensity and for 64 byte object size, the

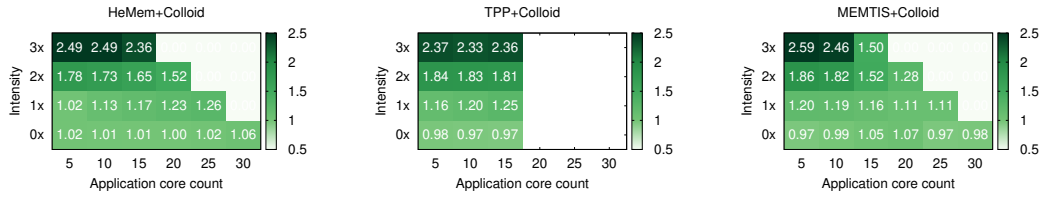



Figure 4.10: Independent of the number of application cores, Colloid continues to improve performance under memory interconnect contention without impacting performance under no memory interconnect contention across all systems (a-c; left-right). Each cell in the heatmap shows performance improvement enabled by Colloid (throughput of the underlying system with Colloid normalized by throughput without Colloid) for a specific number of application cores and memory interconnect contention intensity. For TPP, we found that with 20 or more application cores, page migrations stall; thus we omit these results.

alternate tier bandwidth utilization is 53% of the theoretical maximum bandwidth for this workload; for 4096 byte object size, the alternate tier bandwidth utilization is 96% of the theoretical maximum bandwidth for this workload.

**Impact of application core count.** We vary the number of application cores from 5 – 30. For a given number of application cores ($x$), we run the memory antagonist on a varying number of the remaining cores ($30 - x$) to control memory interconnect contention intensity. We find that Colloid properties continue to hold independent of the number of application cores—across the spectrum, with 1 – 3× memory interconnect contention intensity, Colloid improves performance of HeMem by 1.02 – 2.49×, TPP by 1.16 – 2.37× and MEMTIS by 1.11 – 2.59× without impacting performance for 0× memory interconnect contention intensity case (Figures 4.10(a),4.10(b),4.10(c)). For a fixed number of application cores, the magnitude of performance improvement increases with increasing memory interconnect contention intensity. For a fixed memory interconnect contention intensity, varying the number of application cores results in different trends for different systems.

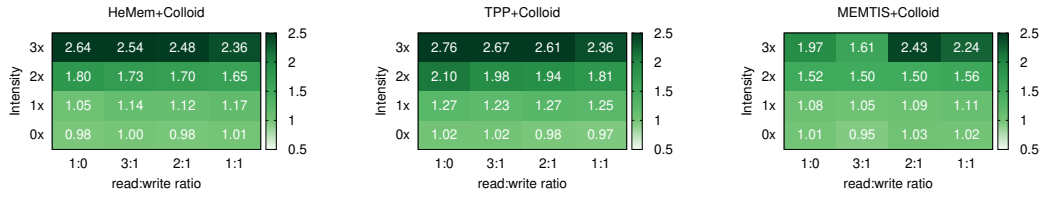



Figure 4.11: Independent of the workload read/write ratio, Colloid continues to improve performance under memory interconnect contention without impacting performance under no memory interconnect contention across all systems (a-c; left-right). Each cell in the heatmap shows performance improvement enabled by Colloid (throughput of the underlying system with Colloid normalized by throughput without Colloid) for a specific read/write ratio and memory interconnect contention intensity.

**Impact of read/write ratio.** We vary the read/write ratio of the GUPS workload from 1 : 0 to 1 : 1 while keeping all other parameters fixed to their default values. Colloid properties continue to hold independent of the read/write ratio—with 1 – 3× memory interconnect contention intensity, Colloid improves performance of HeMem by 1.05 – 2.64×, TPP by 1.23 – 2.76× and MEMTIS by 1.05 – 2.43×, without impacting performance for 0× memory interconnect contention intensity. At 1× memory interconnect contention intensity, there is little variation in Colloid performance improvement, across all systems; for 2× and 3× memory interconnect contention intensity, Colloid performance improvement reduces with increasing fraction of writes for HeMem and TPP, and does not exhibit a clear trend for MEMTIS.

**Colloid CPU overheads.** Colloid requires additional CPU cycles for latency measurements and page placement algorithm. Measurements for Figure 4.5 experiments for each system without and with Colloid suggest that Colloid has <2% CPU overheads for HeMem and MEMTIS, independent of the memory interconnect contention intensity. Colloid has slightly higher CPU overheads (4 – 6.5%) for TPP due to an additional core being used for access latency measurement (§4.4).

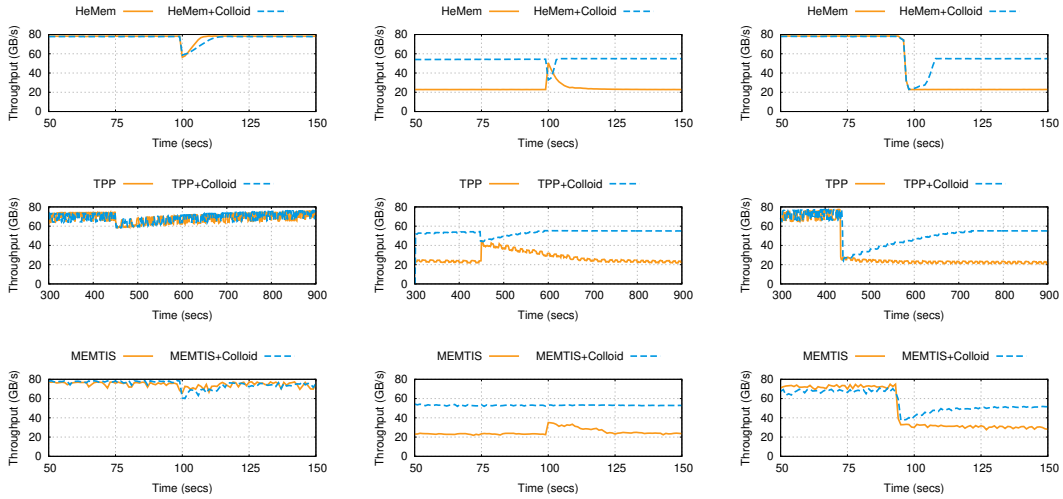



Figure 4.12: Colloid enables the underlying system to maintain its convergence time upon dynamic changes in access pattern and/or memory interconnect contention intensity. Each row (top-bottom) shows instantaneous throughput for the system, measured at per-second timescale, with and without Colloid. Each column uses a different scenario: (left) change in access pattern under no memory interconnect contention (center) change in access pattern under memory interconnect contention (right) change in memory interconnect contention intensity.

Adapting to dynamic, time-varying, workloads is a common challenge for any memory tiering system—if the workload changes faster than the tiering system can adapt, then the effectiveness of the memory tiering system reduces. Fundamentally, there are two sources of dynamism: change in memory access pattern, and change in memory interconnect contention. We evaluate the impact of Colloid on the convergence time of the underlying system to each of these separately.

**Dynamism due to change in access pattern.** We dynamically change the hot set of the GUPS workload using the same methodology as in HeMem [185]. We run the application on 15 cores with 0× memory interconnect contention intensity and allow enough time for each of the systems to reach steady-state. Then, at a particular instant of time ($t = 100$), we instantaneously change the workload's hot set—pages previously in the hot set are marked as cold, and a

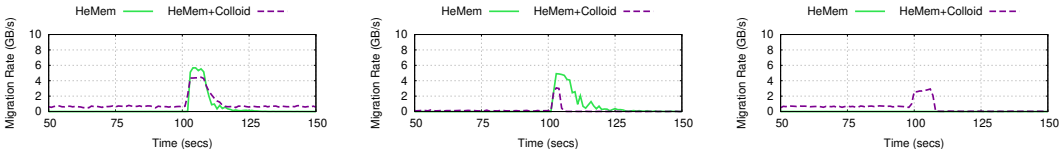



Figure 4.13: Migration rate (measured per-second) over time for HeMem and HeMem+Colloid corresponding to Figure 4.12 experiments.

different set of random pages are marked as hot. Figure 4.12 (top, left) shows the result with HeMem. Before the change, both HeMem and HeMem+Colloid are operating at the same throughput—both of them place the entire hot set in default tier. Upon the change, there is a sudden drop in throughput since several pages in the new hot set are now in the alternate tier. Both HeMem and HeMem+Colloid detect the change in access distribution, migrate the new hot pages to default tier, and converge back to the original throughput. We observe similar trends for TPP and MEMTIS, as shown in Figure 4.12 (center, left) and (bottom, left). TPP takes significantly longer relative to HeMem and MEMTIS (100s of seconds) to converge after change in access pattern because it takes longer to identify the new hot pages due to page table scan latency and less precise access information; Colloid does not change TPP's convergence times.

Figure 4.13 (left) shows the migration rate (that is, the number of bytes migrated per second) for HeMem and HeMem+Colloid over time for the above experiment. As expected, both HeMem and HeMem+Colloid observe sudden increase in migration rate upon change of the workload. However, compared to HeMem, HeMem+Colloid migration rate decreases more gradually; this is because of the dynamic migration limit in the Colloid algorithm (§4.3.3)—as the system approaches the equilibrium point, smaller $\Delta p$ values lead to lower migration rates. Overall, this relatively more gradual decrease in migration rate results in HeMem+Colloid taking marginally longer than HeMem (~3 seconds)

to converge. This overhead does not scale with increasing convergence time; for example, if the hot set size is larger, HeMem+Colloid migration rate will be bound by the static migration limit for most of the time; the dynamic migration limit will apply only when $\Delta p$ becomes relatively small at which point Colloid is already close to convergence. Overall, HeMem+Colloid does not exceed HeMem's peak migration rate. In the steady state, HeMem+Colloid has marginally higher migration rate compared to HeMem, but this does not impact application performance since it is very small (<0.7% of application throughput). For TPP and MEMTIS peak migration rate is < 1.6% and 4.5% of application throughput, respectively.

We repeat the same experiment as above, but with 3× memory interconnect contention intensity, to understand the behavior of the systems with change in access distribution under memory interconnect contention. Figure 4.12 (top, center) shows that, before the change, HeMem+Colloid operates at higher throughput by placing part of the hot set in the alternate tier. Upon the change, HeMem throughput increases since pages in the new hot set are now in the alternate tier; HeMem detects the change and migrates all hot pages back to the default tier thus converging back to its original suboptimal throughput. On the other hand, upon the change, HeMem+Colloid throughput degrades since larger-than-ideal fraction of the new hot set is in the default tier. HeMem+Colloid detects the change and recovers back to its original throughput by migrating pages to maintain the ideal fraction of hot set in the alternate tier. Both HeMem and HeMem+Colloid converge within 10 seconds. We observe a similar trend for TPP, although at a different timescale. MEMTIS+Colloid throughput is not impacted by the change. This is because MEMTIS proactively places cold pages in the alternate tier even when there is

capacity available in the default tier. As a result, for MEMTIS+Colloid, the entire working set is already in the alternate tier before the change.

**Dynamism due to change in memory interconnect contention.** We keep the workload access distribution fixed, and dynamically change the intensity of memory interconnect contention. Initially, the application is run on 15 cores with 0× memory interconnect contention. At a particular time instant, we introduce 3× memory interconnect contention. Figure 4.12 (top, right) shows that, before the change, both HeMem and HeMem+Colloid operate at same throughput with the entire hot set in default tier. Upon the change, there is a sudden drop in throughput due to increased memory interconnect contention. HeMem, being agnostic to memory interconnect contention, does not react and continues to operate at degraded throughput. HeMem+Colloid detects the change, migrates pages in the hot set to alternate tier, and converges to a point of higher throughput within ~10 seconds (similar timescale as its reaction to access pattern changes). The throughput that it converges to closely matches the best-case throughput for 3× memory interconnect contention (Figure 4.5). We observe similar trends for TPP and MEMTIS, as shown in Figure 4.12—both of them converge to near best-case throughput at similar timescales at which they adapt to access pattern changes (~250s and ~25s, respectively).

We now evaluate Colloid benefits to end-to-end performance of three real-world applications that exhibit different compute-to-memory bandwidth demands and access patterns. As before, we run each application on 15 cores and evaluate with varying intensity of memory interconnect contention.

**Graph processing (GAPBS).** This application implements several graph processing algorithms operating on top of in-memory graphs, producing a large

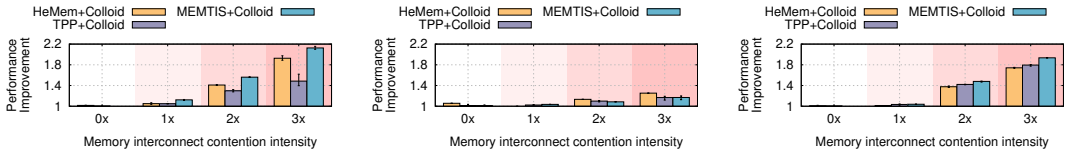



Figure 4.14: Colloid enables end-to-end performance benefits for real-world applications that exhibit different compute-to-memory bandwidth demands and access patterns (a-c; left-right). (a) GAPBS, a graph processing application; (b) Silo, an in-memory transactional database; and, (c) CacheLib, an in-memory key-value cache.

number of memory accesses in the process. We evaluate the PageRank algorithm in GAPBS [26] on a Twitter graph, similar to [132]. Access locality arises from skew in the degree distribution of graph nodes. We set the default tier size to one-third of the total working set size (~12.6GB) and execute 16 trials of the algorithm. The performance metric is the average execution time across all trials (lower is better). Figure 4.14(a) shows that, with increasing memory interconnect contention, Colloid improves the performance of HeMem by $1.05\times - 1.92\times$, TPP by $1.05 - 1.48\times$ and MEMTIS by $1.12 - 2.12\times$. Performance improvement with TPP is relatively smaller compared to HeMem and MEMTIS because it takes longer to identify hot pages.

**In-memory transaction processing (Silo).** This application implements a high-performance in-memory transactional database; we evaluate it using the YCSB-C benchmark, as in [132]. The working set consists of 400 million key-value pairs with 64 byte keys and 100 byte values; the total working set size is thus ~60GB. Default tier size is set to one-third of the working set size. We execute 15 billion lookup operations using a Zipfian access distribution. Figure 4.14(b) shows the corresponding results. Colloid matches baseline performance at low memory interconnect contention and improves the performance of HeMem by $1.13 - 1.25\times$, TPP by $1.09 - 1.17\times$ and MEMTIS by $1.08 - 1.17\times$, at higher memory interconnect contention intensities.

**In-memory key-value cache (CacheLib).** We evaluate CacheLib, a popular open-source caching library, in RAM-only mode and execute workloads using the standard CacheBench tool. We use the key-value workload from [185] (HeMemKV): the key and the value sizes are fixed to 64B and 4KB, respectively, 20% of keys are in the hot set, and remaining are in the cold set. The hot set is accessed uniformly at random with 90% probability, and cold set with 10% probability. The GET/UPDATE ratio is 90/10. We populate 15 million KV pairs leading to working set size of ~75GB, set the default tier size to one-third of the working set size, and execute 1 billion operations from 15 cores. The performance metric is average throughput (operations per second) over the entire duration of the experiment. Figure 4.14(c) shows that, at 2× and 3× memory interconnect contention intensity, Colloid improves performance of HeMem by 1.37 – 1.74×, TPP by 1.42 – 1.79×, and MEMTIS by 1.48 – 1.93×.

## 4.6 Related work

We discuss key works related to Colloid.

**Memory management for NUMA architectures.** Classical literature in this space [32, 33, 58, 133, 147, 212] has primarily focused on the problem of placing both compute tasks and memory pages in multi-socket systems to optimize application performance. The work that comes closest to our problem is Carrefour [58]. Carrefour, among other techniques, performs page placement to balance load (average rate of requests) across memory attached to different NUMA nodes. In the memory tiering context, where each tier has a different unloaded latency, this will lead to unnecessarily placing pages in alternate tiers

even when the memory interconnect is not contended. Moreover, under memory interconnect contention, balancing request rates could lead to suboptimal application performance. Colloid demonstrates that memory management using the principle of balancing access latencies is a better approach to maximize application performance.

**Software-managed tiered memory management.** We have already demonstrated the benefits of Colloid with state-of-the-art software-based memory tiering systems [132,150,185]. Colloid can be integrated with any of the other existing systems [2,47,63,115,119,230,233] to perform memory management using the principle of balancing access latencies.

BATMAN [47] performs tiered memory management using a bandwidth-centric approach—it balances fractions of accesses to the tiers based on the ratio of their theoretical maximum bandwidths, independent of memory interconnect contention. Such an approach is suboptimal due to two reasons. First, when unloaded latencies of the tiers are different (as is the case for modern tiered memory architectures), this approach may unnecessarily place hot pages in tiers with higher access latency leading to suboptimal application performance. Second, as discussed in §4.3.1, access latency of tiers can increase significantly, far before their maximum bandwidth is saturated; as a result, using bandwidth as a metric fails to fully capture the impact of memory interconnect contention. Colloid automatically captures unloaded latencies and memory interconnect contention (independent of whether memory bandwidth is saturated or not) using the principle of balancing access latencies.

**Hardware-managed tiered memory management.** An alternate approach to tiered memory management is to use the default tier as an inclusive cache (*e.g.*,

Intel memory mode [134, 185], stacked DRAM caches [87, 113, 143, 183, 199]) or exclusive cache (*e.g.*, Intel flat memory mode [246]) for the alternate tier. These approaches have the benefit of enabling data placement across tiers at cache-line granularity. However, existing hardware-managed tiered memory systems make the same assumption as their software counterparts: despite serving most of the hot data from the default tier, the access latency of the default tier remains lower than that of the alternate tier. It would be interesting to integrate Colloid with hardware-based tiered memory management mechanisms to perform data placement at cacheline granularity using the principle of balancing access latencies.

## 4.7 Implications for the rest of the thesis

Existing operating system memory management mechanisms innovate on access tracking, page migration, and dynamic page size determination; however, they all use the same page placement algorithm—packing the hottest pages in local memory and placing the remaining pages in disaggregated memory. This is based on the implicit assumption that, despite serving the hottest pages, access latency of the local memory is always lower than that of disaggregated memory. Building on Chapters 2 and 3, we have demonstrated that this assumption does not hold in the regime of multiple concurrent memory requests leading to memory interconnect contention, and that existing systems achieve far from optimal performance in this regime. We have presented Colloid, a memory management mechanism that embodies the principle that page placement should be adapted to balance the access latencies of local and disaggregated memory. This principle provides a unified approach to memory manage-

ment, enabling Colloid to optimize application performance independent of the latency and bandwidth characteristics of the individual interconnects used for memory disaggregation. We have integrated Colloid with three state-of-the-art memory management systems, and demonstrated its effectiveness over a wide variety of workloads and real-world applications.

In Chapter 3, we demonstrated that individual application performance can improve when memory is disaggregated over cache-coherent interconnects like CXL. The design of Colloid shows how to re-think operating system memory management to actually achieve such operating points, without having to modify applications or manually control their data placement. Beyond maximizing individual application performance, realizing the improved utilization promise of memory disaggregation requires sharing disaggregated memory resources across multiple applications or hosts or tenants. This is the problem that we consider in Chapter 5.

CHAPTER 5

# KARMA: SHARING DISAGGREGATED MEMORY ACROSS TENANTS UNDER DYNAMIC DEMANDS

In this chapter, we first demonstrate that dynamic resource demands are the norm in real-world deployments (§5.2), and show that the classical max-min fairness algorithm for resource allocation loses one or more of its properties under dynamic demands (§5.3). Next, we present the Karma and its theoretical analysis (§5.4). We then discuss Karma's integration with existing disaggregated memory systems (§5.5) and its empirical evaluation (§5.6). Finally, we discuss related work (§5.7), and the broader implications of this chapter (§5.8).

## 5.1 Overview

Sharing disaggregated memory capacity across multiple hosts (or applications or tenants) is fundamentally a resource allocation problem—a resource with a fixed capacity is allocated to one or more "users", where each user is an entity consuming the resource (*e.g.*, host, application, tenant). There is a large and active body of research on designing resource allocation mechanisms that achieve Pareto efficiency (high resource utilization) and strategy-proofness (selfish users should not be able to benefit by lying about their demands) while ensuring that resources are allocated fairly among users, *e.g.*, [78, 83, 106, 179, 181, 198, 201].

For a system containing a single resource, the two most popular allocation mechanisms are strict partitioning [20, 219] and max-min fairness [78, 83, 91, 111, 164, 165, 179, 181, 198]. The former allocates the resource equally across all users ("fair share"), independent of their demands; this guarantees strategy-

proofness and fairness, but not Pareto efficiency since resources can be underutilized when one or more users have demands lower than the fair share. Max-min fairness alleviates limitations of strict partitioning by taking user demands into account: it maximizes the minimum allocation across users while ensuring that each user's allocation is no more than their demand. A classical result shows that resource allocation based on max-min fairness guarantees each of the three desirable properties—Pareto efficiency, strategy-proofness, and fairness. These powerful properties have, over decades, motivated efforts in both systems and theory communities on generalizations of max-min fairness for allocating multiple resources [78, 82, 83], for incorporating application performance goals and deadlines [82, 106, 145, 146], and for new models of resource allocation [48, 56, 68, 84, 181, 198], to name a few.

This chapter explores a complementary problem—resource allocation of a single elastic resource in a system where user demands are dynamic, that is, vary over time. Dynamic user demands are the norm in most real-world deployments [29, 44, 118, 144, 188, 192, 213, 219, 234]; for instance, analysis of production workloads in §5.2 reveals that user demands vary by as much as 17× within minutes, with a majority of users having demands with standard deviation 0.5 – 43× of the average over time. We show in §5.3 that, for systems with such dynamic user demands, resource allocation based on the max-min fairness algorithm fails to guarantee one or more of its properties: (1) if the allocation is done based on demands at $t = 0$, Pareto efficiency and strategy-proofness are no longer guaranteed; and, (2) if the allocation is done periodically, *long-term* fairness is no longer guaranteed—for $n$ users with the same average demand, the max-min fairness algorithm may allocate some user as much as $\Omega(n)$ more resources than other users over time.

We present Karma, a new resource allocation mechanism for dynamic user demands. The key technical contribution of Karma is a credit-based resource allocation algorithm: in each quantum, users receive credits when they donate a part of their fair share of resources (*e.g.*, if their demand is less than their fair share); users can use these credits to borrow resources in any future quantum when their demand is higher than their fair share. When the supply of resources from donors is equal to the demand from borrowers, it is easy to exchange resources and credits among users. The key algorithmic challenge that Karma resolves is when supply is not equal to demand—in such scenarios, Karma carefully orchestrates resources and credits between donors and borrowers: donors are prioritized so as to keep credits across users as balanced as possible, and borrowers are prioritized so as to keep the resource allocation as fair as possible.

We theoretically establish Karma guarantees for dynamic user demands. Karma guarantees Pareto efficiency at all times: in each quantum, it allocates resources such that it is not possible to increase the allocation of a user without decreasing the allocation of at least another user. For strategy-proofness, Karma guarantees that a selfish user cannot increase their aggregate resource allocation by *over*-reporting their demands in any quantum. In addition, we show a new surprising phenomenon (that may primarily be of theoretical interest): if a user had perfect knowledge about the future demands of all other users, the user can increase its own aggregate allocation by a small constant factor by *under*-reporting its demand in some quanta; however, for $n$ users, imprecision in this future knowledge could lead to the user losing $\Omega(n)$ factor of their aggregate resource allocation by under-reporting their demand in any quantum. Put together, these results enable Karma to provide powerful guarantees related to

strategy-proofness. Finally, for fairness, we prove that given a set of (past) allocations, Karma guarantees an optimally-fair resource allocation. We also establish that Karma guarantees similar properties even when multiple selfish users can collude, and even when different users have different fair shares.

We have realized Karma on top of Jiffy [118], an open-sourced multi-tenant disaggregated memory system; an end-to-end implementation of Karma is available at `https://github.com/resource-disaggregation/karma`. Evaluation of Karma over production workloads demonstrates that Karma's theoretical guarantees translate well into practice: it matches the max-min fairness algorithm in terms of resource utilization, while significantly improving the long-term fairness of resources allocated across users. Karma's fairer resource allocation directly translates to application-level performance; for instance, over evaluated workloads, Karma keeps the *average* performance (across users) the same as the max-min fairness algorithm, while reducing performance *disparity* across users by as much as ~2.4×. Karma also incentivizes users to share resources: our evaluation shows that (1) Karma-conformant users achieve much more desirable allocation and performance compared to users who prefer a dedicated fair share of resources; and, (2) if users were to turn Karma-conformant, they can improve their performance by better matching their allocations with their demands over time.

## 5.2 Dynamic demands

Increasingly many applications running data analytics or key-value caches operate on data collected from social media, application and network logs, mobile

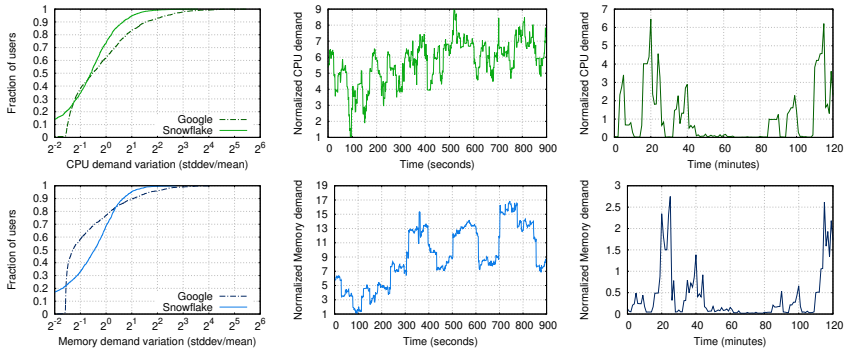



Figure 5.1: Analysis of Google and Snowflake workloads suggests that a large fraction of users have dynamic demands (left) that can change dramatically over short timescales (center, right). (left) CDFs, across users, of the ratio of standard deviation and mean of each user's demand. (center) For a randomly sampled user in the Snowflake trace, the variation in the user's CPU and memory demands (normalized by minimum demand) over a 15 minute period. (right) For a randomly sampled user in the Google trace, the variation in the user's CPU and memory demands (normalized by minimum demand) over a 2 hour period.

systems, etc. A unique characteristic of these data is that they are less controllable by the organization because they are generated by entities outside of the organization. As a result, applications can observe highly time-varying dynamic resource demands [29, 44, 118, 144, 188, 192, 213, 219, 234].

To build a deeper understanding of variation in user demands over time, we analyze two publicly-available production workloads: (1) Google [188] resource usage information across 8 clusters (1000 – 2000 users per cluster) over a 30 day period; and, (2) Snowflake [219], a cloud-based database query engine that provides resource usage statistics for over 2000 users over a 14 day period. To characterize user demand variability over time, we compute—for each user—the ratio of the standard deviation and mean of their demands over the entire period. Figure 5.1 (left) shows that 40 – 70% of all users in both Google and Snowflake workloads have a standard deviation in CPU and memory demands at least 0.5× their mean, indicating high variability in demands for most users. Furthermore, the standard deviation in demands of as many as 20% of the users

can be as high as their mean demand, with some users having extremely high variance in demands (standard deviations up to $12-43\times$ the mean). Similar observations have been made for time-varying user demands in inter-datacenter networks; for instance, production studies [1] show that, on average, user demands vary by 35% within 5-minute intervals, with some demands varying by as much as 45% within a short period of time.

Figure 5.1 (center) shows the CPU and memory demands for a randomly-sampled user from the Snowflake trace over a 15 minute window (we show only one user and only 15 minute window for clarity; analyzing a sample of 100 users, we find 87% of the users to have similar demand patterns). The figure shows that user demands can change dramatically over tens of seconds, by as much as $6\times$ and $2\times$ for compute and memory, respectively. Similarly, we see significant variation in demands even for a random user from the Google trace (shown in Figure 5.1 (right)).

## 5.3 Max-min fairness properties under dynamic demands

The classical max-min fairness algorithm for resource allocation provides many desirable properties, *e.g.*, Pareto efficiency, strategy-proofness, and fairness. However, buried under the proofs is the assumption that user demands are static over time, an assumption that does not hold in practice (as demonstrated in Figure 5.1). For the realistic case of dynamic user demands, max-min fairness can be applied in two ways, each of which leads to violating one or more of its properties. We will demonstrate this using the example in Figure 5.2; here, time is divided into five quanta and three users have demands varying across

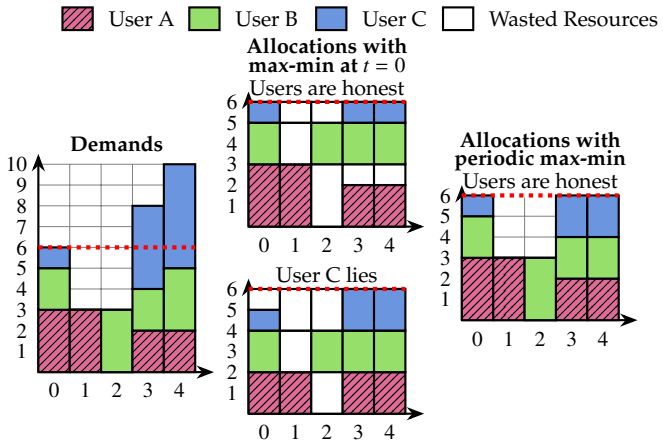



Figure 5.2: Classical max-min fairness guarantees break for dynamic user demands. Here, 6 units of a resource are shared by 3 users (fair share of 2). Discussion in §5.3.

quanta.

First, one can naïvely perform max-min fair allocation just once based on user demands at quantum $t = 0$. This results in max-min fairness losing both Pareto efficiency and strategy-proofness. In the example of Figure 5.2, since allocations will only be done based on the demands specified by the users at $t = 0$, if users were to specify their true demands, user C will obtain an allocation of 1 unit leading to a total useful allocation of 3 units over the entire duration (as shown in Figure 5.2 (middle, top)); if user C were to lie and over-report their demand at $t = 0$ as 2 units, then they can achieve a more desirable total useful allocation of 5 units (Figure 5.2 (middle, bottom)). This breaks strategy-proofness. In addition, max-min fairness is also not Pareto efficient: for many quanta, resources allocated to users will be underutilized as is evident in Figure 5.2 (middle).

A better way to apply max-min fairness for dynamic user demands is to periodically reallocate resources based on users' instantaneous demands (*e.g.*, every quantum of time periods, as in several operating systems and hypervisors [114,

220]). This trivially guarantees Pareto efficiency and strategy-proofness but results in extremely unfair allocation across users. Figure 5.2 (right, top) shows an example where max-min fairness can result in 2× disparity between resources allocated to users over the 5 quanta—user A receives a total allocation of 10 slices, while user C receives a total allocation of only 5 slices, despite them having the same average demand; this example can be easily extended to demonstrate that max-min fairness can, for $n$ users, result in resource allocations where some user gets a factor of $\Omega(n)$ larger amount of resources than other users (proof in Theorem 1 below). Such disparity in resource allocations also leads to disparity in application-level performance across users since, as discussed above in use cases, many applications require consistently good performance over long periods of time, rather than excellent performance at some times and very poor performance at other times [56, 76, 83, 209]. We will demonstrate, in the evaluation section, that users experience significant disparity in application-level performance due to such disparate resource allocations.

**Theorem 1.** *There is an instance with n users where the ratio of the maximum over the minimum total resource allocation in classical max-min fairness is* $n + 1$.

*Proof.* The amount of available resources every quantum is $n$, the fair share of every user is $f = 1$, and there are $n + 1$ quanta in total. The first $n - 1$ users constantly have demand 1 and the final user, A, has 0 demand the first $n$ quanta, but demand $n$ on the final quantum. Classic max-min fairness will always allocate 1 resource to every one of the $n - 1$ users, leading to a total allocation of $n + 1$ in the final quantum, while on the same quantum user A will have received a total allocation of 1. In this scenario, the ratio of maximum over minimum allocation is $n + 1$. □

For the rest of the chapter, we focus on long-term fairness; informally, an allocation is considered fair if all users have the same aggregate resource allocation over time. Our goal is to design a resource allocation mechanism that, for dynamic user demands, guarantees Pareto efficiency, strategy-proofness, and fairness.

## 5.4 Karma

Karma is a resource allocation mechanism for dynamic user demands. Karma uses *credits* (§5.4.1, §5.4.2)—users receive credits when they donate a part of their fair share of resources (*e.g.*, when their demand is less than their fair share), and can use these credits to borrow resources beyond their fair share during periods of high demand. Karma carefully orchestrates the exchange of resources and credits between donors and borrowers: donors are prioritized in a manner that ensures credit distribution across users remains as balanced as possible, and borrowers are prioritized in a manner that keeps the resource allocation as fair as possible. We will prove theoretically in §5.4.3 that, while simple in hindsight, this allocation mechanism simultaneously achieves Pareto efficiency, strategy-proofness, and fairness for dynamic user demands.

### 5.4.1 Preliminaries

We consider the following setup for the problem: we have $n$ users sharing a single resource (CPU, memory, GPUs, etc.); each user has a fair share of $f$ resource units (each unit is referred to as a *slice*), and thus the pool has $n \times f$ slices of

the resource. Time is divided into quanta, users demand a certain number of resource slices every quantum, and Karma performs resource (re)allocation at the beginning of each quantum. While user demands during each quantum can be arbitrary, unsatisfied demands in one quantum do not carry over to the next. Similar to prior work [78, 179, 181, 198], we assume that users are not adversarial (that is, do not lie about their demands simply to hurt others' allocations), but are otherwise selfish and strategic (willing to misreport their demands to maximize their allocations).

### 5.4.2 Karma design

Let $0 \leq \alpha \leq 1$ be a parameter. Karma guarantees that each user is allocated an $\alpha$ fraction of its fair share ($= \alpha \cdot f$) in each quantum; we refer to this as the guaranteed share. Karma maintains a pool of resource slices—karmaPool—that, at any point in time, contains two types of slices:

- **Shared slices** are the slices in the resource pool that are not guaranteed to any user. It is easy to see that the number of shared slices in the system is $n \cdot f - n \cdot \alpha \cdot f = n \cdot (1 - \alpha) \cdot f$.
- **Donated slices**, that are donated by users whose demands are smaller than their guaranteed share.

We use these two sets of slices in the following manner. In any given quantum, if a user has demand less than its guaranteed share, then the user is said to be "donating" as many slices as the difference between the user's guaranteed share and demand in that quantum. A user that has demand larger than its guaranteed share is said to be "borrowing" slices beyond its guaranteed share,

which the system can potentially supply using either shared slices or donated slices.

**Karma credits**

Karma allocates resources not just based on users' instantaneous demands, but also based on their past allocations. To maintain past user allocation information, Karma uses credits.

Users earn credits in three ways. First, each user is bootstrapped with a fixed number of initial credits upon joining the system (we will discuss the precise number shortly, once enough context is provided); second, each user is allocated $(1 - \alpha) \cdot f$ free credits every quantum as compensation for contributing $(1 - \alpha)$ fraction of its fair share to shared slices. Finally, users earn one credit when some other user borrows one of their *donated* slices (one credit per quantum per slice).

Unlike earning credits, there is only one way for any user to lose credits: for every slice borrowed from the karmaPool (donated or shared), the user loses one credit.

**Prioritized resource allocation**

We now describe Karma's resource allocation algorithm, that orchestrates resources and credits across users (Algorithm 3). To make the discussion succinct, we refer to the sum of user demands beyond their guaranteed share as "borrower demand"; that is, to compute borrower demand for any given quantum,

**Algorithm 3 : Karma resource allocation algorithm.**

`demand[u]`: demand of user u in the current quantum
`credits[u]`: credits of user u in the current quantum
`alloc[u]`: allocation of user u in the current quantum
$f$: fair share
$\alpha$: guaranteed fraction of fair share

Every quantum do:

1: `shared_slices` $\leftarrow n \cdot (1-\alpha) \cdot f$
2: For each user u,
3: increment `credits[u]` by $(1-\alpha) \cdot f$
4: `donated_slices[u]` = `max` $(0, \alpha \cdot f -$ `demand[u]`$)$
5: `alloc[u]` = `min` (`demand[u]`, $\alpha \cdot f$)
6: `donors` $\leftarrow$ all users u with `donated_slices[u]` $> 0$
7: `borrowers` $\leftarrow$ all users u with
8: `alloc[u]`<`demand[u]` & `credits[u]`>0
9: **while** `borrowers` $\neq \phi$ and
10: $(\sum_u$ `donated_slices[u]` $> 0$ or `shared_slices` $> 0)$ **do**
11: $b^\star \leftarrow$ borrower with maximum `credits`
12: **if** `donors` $\neq \phi$ **then**
13: $d^\star \leftarrow$ donor with minimum `credits`
14: Increment `credits`[$d^\star$] by 1
15: Decrement `donated_slices[u]` by 1
16: Update the set of `donors` (line 6)
17: **else**
18: Decrement `shared_slices` by 1
19: Increment `alloc`[$b^\star$] by 1
20: Decrement `credits`[$b^\star$] by 1
21: Update the set of `borrowers` (line 7)

we take all users with demand greater than their guaranteed share and sum up the difference between their demand (in that quantum) and $\alpha \cdot f$. In quanta when borrower demand is equal to the supply (number of slices in karmaPool), Karma's decision-making is trivial: simply allocate all slices in karmaPool to the borrowers, and update credits for all users as described in the previous subsection. The key algorithmic challenge that Karma resolves is when the supply is either more or less than the borrower demand. We describe Karma allocation mechanism for such scenarios next and then provide an illustrative example.

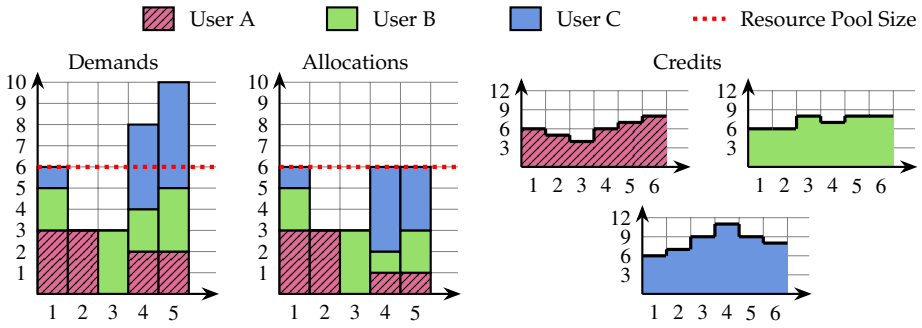



Figure 5.3: Karma resource allocation for the running example of Figure 5.2. Recall that there are 6 resource slices, 3 users each with average demand and fair share equal to 2. We show the case of the guaranteed share being 1 ($\alpha = 0.5$), with 6 bootstrapping (initial) credits for each user. Note that each user receives 1 free credit every quantum. Karma achieves significantly improved fair allocation than max-min fairness—it allocates each user an equal allocation of 8 resource slices over time.

**Orchestrating resources and credits when supply > borrower demand.** When supply is greater than borrower demand, there are enough slices in karmaPool to satisfy the demands of all borrowers. In such a case, Karma prioritizes the allocation of donated slices over shared slices (so that donors get credits), and across multiple donated slices, prioritizes the allocation of a slice from the donor that has the smallest number of credits—this allows "poorer" donors to earn more credits, and moves the system towards a more balanced distribution of credits across users. Intuitively, credits capture the allocation obtained by a user until the last quantum—users who obtained lower allocations in the past will have a higher than average (across users) number of credits, while those who received a surplus of allocations will have a below-average number of credits. Hence, balancing the number of credits across users over time allows Karma to move towards a more equitable set of total allocations across users. Once all donated slices are allocated, Karma allocates shared slices to satisfy the remaining borrower demands.

**Orchestrating resources and credits when supply < borrower demand.** When

supply is less than demand, karmaPool does not have enough slices to satisfy all borrower demands. In such a scenario, Karma prioritizes allocating slices to users with the maximum number of credits. This strategy essentially favors users that had fewer allocations in the past (and thus, a larger number of credits), hence moving the system towards a more balanced allocation of resources across users, promoting fairness. At the same time, reducing the credits for the users with the most credits also moves the system to a more balanced distribution of credits across users.

**Illustrative example.** The running example in Figure 5.3 shows the execution of Karma's algorithm for the example from Figure 5.2 for $\alpha = 0.5$: that is three users A, B, and C, each with a fair share 2 slices ($f = 2$), and a guaranteed share of 1 slice. Recall that, since $(1 - \alpha) \cdot f = 1$, each user receives 1 credit every quantum, and suppose all users are bootstrapped with 6 initial credits.

In the first quantum, C's demand is equal to the guaranteed share, while A and B request 2 and 1 slices beyond the guaranteed share, respectively. Since supply (= 3 shared slices in karmaPool) is equal to borrower demand, Karma uses the shared slices to allocate slices beyond the guaranteed share for A and B and satisfies their demands. This results in a final allocation of 3 slices for $A$, 2 slices for $B$, and 1 slice for $C$. $A$ loses 2 credits, $B$ loses 1 credit and no one gains any credits.

In the second quantum, A demands 3 slices, while B and C donate 1 slice each. The total supply (= 5, with 2 donated slices and 3 shared slices) exceeds the borrower demand. A is allocated 3 slices and it loses 3 credits (since its allocation is 2 slices above its guaranteed share). B and C receive 1 credit each since their donated slices are used. Similarly, in the third quantum, B demands

3 slices, while A and C donate 1 slice each. Since total supply exceeds borrower demand, B receives the 3 slices it asked for, and loses 2 credits; A and C gain 1 credit each.

The fourth quantum is important: here, demand exceeds supply, and there are no donated slices. Now, unlike classic max-min fairness, Karma will prioritize the allocation of resources based on the credits of each tenant. Since at the start of this quantum, C has 11 credits, while A and B have only 6 and 7 credits respectively, C will be able to get 3 extra slices from the pool of shared slices by using 3 credits and achieve an allocation of 4. A and B will get their guaranteed allocation of 1 and do not gain or lose any credits.

In the fifth quantum, once again, demand exceeds supply. C has 9 credits, B has 8 credits, and A has 7 credits. Karma first prioritizes allocating to C giving it 1 extra slice, at which point both C and B have equal credits (8). Next, they both get 1 extra slice each, at which point the supply is exhausted. The final resulting allocation is 1 slice for A, 2 slices for B, and 3 slices for C.

In the end, A, B, and C end up with the exact same total allocation (8 slices) and number of credits (unlike max-min fairness where user allocations had a disparity of 2×).

We now briefly discuss some additional aspects of Karma design not included in the above description.

**Bootstrapping Karma with initial credits.** Recall that, to bootstrap users, Karma allocates each user an initial number of credits. The precise number of initial credits has little impact on Karma's behavior; after all, credits in Karma essentially capture a relative ordering between users, rather than having any

absolute meaning. The only importance of the number of credits is to ensure that no user runs out of credits at any quantum (which, in turn, could lead to violation of Karma's Pareto efficiency guarantees): even if spare resources are available, a user with high demand may not be able to borrow resources beyond the guaranteed share (line 7 of Algorithm 3) due to running out of credits. Thus, Karma sets the number of initial credits to a large numerical value to ensure that no user ever runs out of credits[1].

**User churn.** Fairness is relatively ill-defined when users can join and leave the system on a short-term basis (*e.g.*, when a user runs a short query with large parallelism, and then leaves the cluster). Also, recall from our motivating scenarios, fair resource allocation in private clouds is usually performed for long-running services. However, Karma still handles user churn since, in many realistic scenarios, the set of all users of the system may not be known upfront during system initialization. For users that join and leave over longer timescales, Karma handles user churn with a simple mechanism: its credits. When a new user joins, either the resource pool size remains fixed and the fair share of all users is reduced proportionally or the resource pool size increases and the fair share of users remains the same. The credits of the existing $n-1$ users do not change, and the new user is bootstrapped with initial credits equal to the current average number of credits across the existing $n-1$ users. Intuitively, users who have donated more resources than they have borrowed will have above-average credits, and those who have borrowed more than they have donated will have below-average credits. As such, initializing the new user with the average number of

[1]For example, in a system with 100 users with fair share of 100 slices, setting initial credits to say $10^{13}$ will ensure that even a worst-case user with highest possible demand (10000 slices) during all quanta cannot run out of credits for $\sim$ 31 years, which is good enough for all practical purposes.

credits (heuristically) puts the new user on equal footing with an existing user that has borrowed and donated equal amounts of resources over time. When a user leaves the system, the fair share of the remaining users is increased proportionally (or resource pool size reduces while maintaining the same fair share), and there is no change in their credits.

**Users with different fair shares.** To generalize the algorithm to users with different fair shares, users with larger weights are charged fewer credits to borrow resources beyond their guaranteed share when compared to users with smaller weights. Intuitively, this enables users with larger weights to obtain more resources than users with smaller weights for the same number of credits. We achieve this by updating Line 20 of Algorithm 3 to decrement credits by $\frac{1}{n \cdot w_i}$ instead of 1, where $w_i$ is the normalized weight of the corresponding user, and $n$ is the number of users. Algorithm 4 presents the full description of the weighted version of Karma's algorithm. For users with different fair shares, as we will discuss in §5.4.3, this generalization leads to the same properties and guarantees, with only a minor difference.

**System parameters, and interpretation for $\alpha$.** Karma has only one parameter: $\alpha$; one can think of resource slice size and quantum duration as parameters, but these are irrelevant to Karma's guarantees: they hold for any slice size and quantum duration, as long as demands change at coarser timescales than the quantum duration. The $\alpha$ parameter in Karma provides a tradeoff between instantaneous and long-term fairness. Providers can choose any $\alpha$ depending on the desired properties. Intuitively, an $\alpha$ smaller than 1 leads to a larger portion of shared slices, giving Karma's algorithm more flexibility in adjusting allocations to achieve better long-term fairness.

**Algorithm 4 : Karma with different fair shares.**

`demand[u]`: demand of user `u` in the current quantum
`credits[u]`: credits of user `u` in the current quantum
`alloc[u]`: allocation of user `u` in the current quantum
$f_u$: fair share of user `u`
$w_u$: normalized weight of user `u` $\left(\frac{f_u}{\sum_k f_k}\right)$
$\alpha$: guaranteed fraction of fair share

Every quantum do:

1: `shared_slices` $\leftarrow (1-\alpha) \cdot (\sum_k f_k)$
2: For each user `u`,
3: increment `credits[u]` by $(1-\alpha) \cdot \frac{\sum_k f_k}{n}$
4: `donated_slices[u]` = `max` $(0, \alpha \cdot f_k -$ `demand[u]`$)$
5: `alloc[u]` = `min` (`demand[u]`, $\alpha \cdot f_k$)
6: `donors` $\leftarrow$ all users `u` with `donated_slices[u]` $> 0$
7: `borrowers` $\leftarrow$ all users `u` with
8: `alloc[u]`<`demand[u]` & `credits[u]`>0
9: **while** `borrowers` $\neq \phi$ and
10: $(\sum_u$ `donated_slices[u]` $> 0$ or `shared_slices` $> 0)$ **do**
11: $b^\star \leftarrow$ borrower with maximum `credits`
12: **if** `donors` $\neq \phi$ **then**
13: $d^\star \leftarrow$ donor with minimum `credits`
14: Increment `credits`$[d^\star]$ by 1
15: Decrement `donated_slices[u]` by 1
16: Update the set of `donors` (line 6)
17: **else**
18: Decrement `shared_slices` by 1
19: Increment user $b^\star$ allocation by 1
20: Decrement `credits`$[b^\star]$ by $\frac{1}{n \cdot w_{b^\star}}$
21: Update the set of `borrowers` (line 7)

### 5.4.3 Karma theoretical properties

In this section, we present a theoretical analysis of Karma. Recall from §5.4.1 that, similar to all prior works, users are considered selfish and strategic (that is, are willing to misreport their demands to maximize their allocations), but not adversarial (that is, do not lie about their demands simply to hurt others' allocations). For the purpose of our theoretical analysis, we assume that Karma is initialized with a large enough number of initial credits so that users do not run

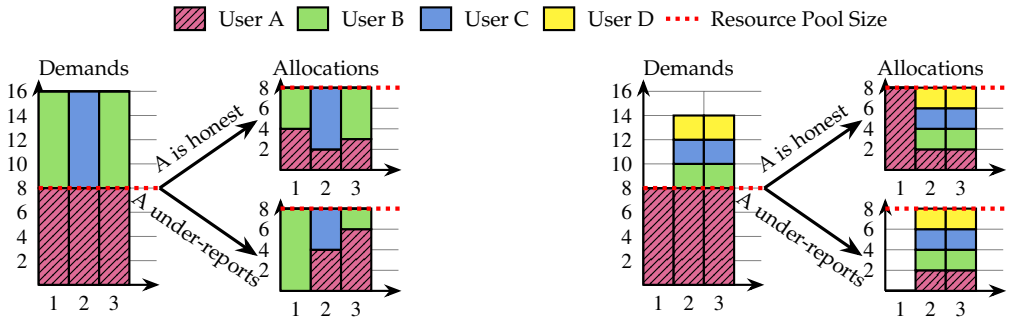



Figure 5.4: The phenomenon of users (left) gaining a small factor of improvement in their allocations by specifying demands less than their real demands, by exploiting knowledge of all future demands of all users; (right) any imprecision in the knowledge of future demands of all users could result in a significant reduction in useful allocations of the lying user. The resource pool has 8 slices, and 4 users with fair share of 2 and guaranteed share of 0 ($\alpha = 0$).

out of credits during the execution of the algorithm (recall, we discussed how to achieve this in practice in § 5.4.2). All our results hold for $\alpha = 0$; extending our results to $\alpha > 0$ is an interesting open question. Here, we present results assuming that all users have equal fair shares. For generalization of our proofs to the case of users with different fair shares, we refer the reader to [69].

We define Pareto efficiency on a per-quantum basis. An allocation is said to be Pareto efficient if it is not possible to increase the allocation of a user without decreasing the allocation of at least one other user by a similar total amount during that quantum. Note that, Pareto efficiency on a per-quantum basis implies Pareto efficiency over time.

**Theorem 2.** *Karma is Pareto efficient.*

Karma's Pareto efficiency follows trivially from the observation that similar to max-min fairness, Karma allocation satisfies the two properties: (1) no user is allocated more resources than its demand, and (2) either all resources are allocated or all demands are satisfied.

For strategy-proofness, we make two important notes. First, if one assumes that the system has a priori knowledge of all future user demands, the resource allocation problem can be solved using dynamic programming; however, for many use cases, it is hard to have a priori knowledge of all future user demands. This leads to our second note: Karma is solving an "online" problem (that is, it does not assume a priori knowledge of future user demands), and thus, we prove online strategy-proofness [8] defined as follows: assume that all users are honest during quanta 0 to $q - 1$; then, a mechanism is said to be online strategy-proof if, for any quantum $q$, a user cannot increase its allocation during quantum $q$ by lying about its demand during quantum $q$.

**Theorem 3.** *Karma is online strategy-proof.*

To prove Theorem 3, we actually prove a stronger result stated below. Karma's online strategy-proofness trivially follows from this.

**Lemma 1.** *A user cannot increase its useful resource allocation by specifying a demand higher than its real demand in any quantum.*

The proof for the lemma is a bit involved, but intuitively, it shows the following. The immediate effect of a user specifying a demand higher than its actual demand is that if the user is allocated more resources than its actual demand, these extra resources do not contribute to its utility, but do put the user into a disadvantageous position: not only can this user lose credits (either because it's asking for resources beyond its guaranteed share, or because it could have gained credits if this extra resource could have been allocated to some borrower), but also because other users get fewer resources; this makes other users be favored by the allocation algorithm in the future while making the lying user

less favored. Thus, the user cannot increase its long-term "useful" allocation by specifying a demand higher than the real demand in any quantum. Specifically, it is possible that when a user over-reports its demand during quantum $q'$, the user receives an increased instantaneous allocation during some future quantum $q > q'$; however, we are able to show that, in this case, the user will also receive reduced instantaneous allocation(s) during other quantum(s) in between $q'$ and $q$, leading to either a lower or equal total allocation over the period between $q'$ and $q$. The hardness in the proof stems from carefully analyzing such cascade effects: a small change in users' resource allocation in any quantum can result in complex changes in future allocations that may lead to higher instantaneous but equal or lower total allocations in future quanta. Once we prove this lemma, the proof for Karma's online strategy-proofness follows immediately.

Before presenting the proof of Lemma 1, we first establish some preliminaries that will be useful for the remainder of this section. We denote $x^+ = \max(x, 0)$. The set of users is denoted with $[n]$. We denote with $r_i^t$ the allocation of user $i$ in quantum $t$. We also denote with $R_i^t$ the cumulative allocation of user $i$ up to quantum $t$, i.e. $R_i^t = \sum_{\tau=1}^{t} r_i^\tau$. By definition, $R_i^0 = 0$. Since we focus on the $\alpha = 0$ case, we note that the number of credits user $i$ earns in quantum $t$ is $f - r_i^t$. Using the notation of this section, this means that Karma prioritizing users with the most credits is equivalent to prioritizing users with the least cumulative allocation. Every quantum $t$, each user $i$'s real demand is $d_i^t$. The useful allocation of user $i$ on quantum $t$ is $u_i^t = \min(r_i^t, d_i^t)$. The total useful allocation of user $i$ after quantum $t$ equals the sum of useful allocations up to that quantum, i.e. $U_i^t = \sum_{\tau=1}^{t} u_i^\tau = \sum_{\tau=1}^{t} \min(r_i^\tau, d_i^\tau)$. Without loss of generality, we are usually going to study the possible deviations of user 1, i.e. how much user 1 can increase its allocation by lying about its demand. We use the symbols $\hat{d}_i^t$, $\hat{r}_i^t$, $\hat{R}_i^t$, $\hat{u}_i^t$, $\hat{U}_i^t$ to

denote the claimed demand and resulting outcome of some deviation of user 1.

We first prove a couple of auxiliary lemmas needed for the proof of Lemma 1.

**Lemma 2.** *Fix a quantum $t$ and let $i$, $j$ be two different users. If the following conditions hold*

- $r_i^t < \hat{r}_i^t$ *and* $r_i^t < d_i^t$*, i.e. user $i$ gets more resources on quantum $t$ when user* 1 *deviates and user $i$ could have gotten more resources when user* 1 *does not deviate.*
- $r_j^t > \hat{r}_j^t$ *and* $\hat{r}_j^t < \hat{d}_j^t$*, i.e. user $j$ gets less resources on quantum $t$ when user* 1 *deviates and user $j$ could have gotten more resources when user* 1 *deviates.*

*then $R_i^t \geq R_j^t$ and $\hat{R}_i^t \leq \hat{R}_j^t$, implying*

$$\hat{R}_i^t - R_i^t \leq \hat{R}_j^t - R_j^t \tag{5.1}$$

It should be noted that the conditions for $i$ (similarly for $j$) can be simplified if $i$ has the same demand in both outcomes (which is trivially true if $i \neq 1$): if $r_i^t < \hat{r}_i^t$ the other inequality is implied as $\hat{d}_i^t = d_i^t$ and $\hat{r}_i^t \leq \hat{d}_i^t$.

*Proof.* Because of the conditions, we notice that $r_i^t < d_i^t$ and $r_j^t > 0$ (the last inequality is true because of $\hat{r}_j^t < \hat{d}_j^t$ which guarantees that $\hat{r}_j^t \geq 0$ and $r_j^t > \hat{r}_j^t$). This implies that it would have been feasible to increase $r_i^t$ by decreasing $r_j^t$. This implies that $R_i^t \geq R_j^t$; otherwise Karma would have prioritized user $j$ and given it some of the resources user $i$ got. With the analogous inverse argument (we can increase $\hat{r}_j^t$ by decreasing $\hat{r}_i^t$) we can prove that $\hat{R}_i^t \leq \hat{R}_j^t$. This completes the proof. □

The main technical tool in our proof is the following lemma bounding the total amount all the users have "won" because of user 1 deviating, i.e. $\sum_k (\hat{R}_k^t -$

$R_k^t)^+$. So rather than bounding the deviating user 1's gain directly, it is better to consider the overall increase in all users combined. More specifically, the lemma upper bounds the increase of that amount after any quantum, given that user 1 does not over-report her demand. The bound on the total over-allocation $\sum_k (\hat{R}_k^t - R_k^t)^+$ then follows by summing over the time periods. The bound on the increase of $\sum_k (\hat{R}_k^t - R_k^t)^+$ after any quantum is different according to three different cases:

- If all users' demands are satisfied, then the increase is at most 0.
- If user 1 is truthful the increase is again at most 0 so in these steps over-allocation can move between users but cannot increase. This is the reason working with the total over-allocation is so helpful.
- If user 1 under-reports the increase is bounded by the amount of resources she receives when she is truthful.

**Lemma 3.** *Fix any $t \geq 1$. Let $\{R_i^{t-1}\}_{i\in[n]}$ and $\{\hat{R}_i^{t-1}\}_{i\in[n]}$ be the cumulative allocations up to quantum $t-1$. Assume that $\{d_i^t\}_{i\in[n]}$ are some users' demands and that $\{\hat{d}_i^t\}_{i\in[n]}$ are the same demands except user 1's, who deviates but does not over-report, i.e. $\hat{d}_1^t \leq d_1^t$. Then it holds that*

$$\sum_{k\in[n]} \left(\hat{R}_k^t - R_k^t\right)^+ - \sum_{k\in[n]} \left(\hat{R}_k^{t-1} - R_k^{t-1}\right)^+ \leq r_1^t \mathbb{1}\left[\hat{d}_1^t < d_1^t\right] \tag{5.2}$$

When all demands are satisfied, user 1 clearly cannot change other users' allocations by under-reporting. We will use Lemma 2 to show that if user 1 is truthful on quantum $t$, then the l.h.s. of (5.2) is at most 0; as the mechanism allocates resources such that the large $\hat{R}_i^t - R_i^t$ are decreased and the small $\hat{R}_i^t - R_i^t$ are increased. Finally, if user 1 under-reports its demand then the (at most)

$r_1^t$ resources user 1 does not get might increase the total over-allocation by the same amount.

*Proof.* Define $P^t = \{i \in [n] : \hat{R}_i^t \geq R_i^t\}$ for all $t$. Suppose by contradiction:

$$\sum_{k \in P^t} \left(\hat{R}_k^t - R_k^t\right) - \sum_{k \in P^{t-1}} \left(\hat{R}_k^{t-1} - R_k^{t-1}\right) > r_1^t \mathbb{1}\left[\hat{d}_1^t < d_1^t\right]$$

Because $\sum_{k \in P^t}(\hat{R}_k^{t-1} - R_k^{t-1}) \leq \sum_{k \in P^{t-1}}(\hat{R}_k^{t-1} - R_k^{t-1})$, the above inequality implies

$$\sum_{k \in P^t} (\hat{r}_k^t - r_k^t) > r_1^t \mathbb{1}\left[\hat{d}_1^t < d_1^t\right] \tag{5.3}$$

Because user 1 does not over-report its demand, it holds that $\sum_k r_k^t \geq \sum_k \hat{r}_k^t$, i.e. the total resources allocated to the users does not increase when user 1 deviates. Combining this fact with (5.3) we get that

$$\sum_{k \notin P^t} (r_k^t - \hat{r}_k^t) > r_1^t \mathbb{1}\left[\hat{d}_1^t < d_1^t\right] \tag{5.4}$$

We notice that because of (5.3), there exists a user $i \in P^t$ for whom $\hat{r}_i^t > r_i^t$; because of (5.4), there exists a user $j \notin P^t$ for whom $r_j^t > \hat{r}_j^t$. Additionally for that $j$ we can assume that $\hat{d}_j^t = d_j^t$ because:

- If user 1 does not deviate then for all $k$, $\hat{d}_k^t = d_k^t$.
- If $\hat{d}_1^t < d_1^t$, then (5.4) implies $\sum_{k \notin P^t, k \neq 1} \left(r_k^t - \hat{r}_k^t\right) > 0$, i.e. $j \neq 1$ and we assumed that only user 1 deviates.

Thus we have $\hat{d}_i^t \leq d_i^t$ (since no user over-reports), $\hat{d}_j^t = d_j^t$, $\hat{r}_i^t > r_i^t$, and $\hat{r}_j^t < r_j^t$. Now Lemma 2 proves that $\hat{R}_i^t - R_i^t \leq \hat{R}_j^t - R_j^t$. This leads to a contradiction, because $i \in P^t$ and $j \notin P^t$, i.e. $\hat{R}_i^t - R_i^t \geq 0 > \hat{R}_j^t - R_j^t$. □

Now we prove Lemma 1.

*Proof of Lemma 1.* Fix a quantum $T$ and let $\{\hat{d}_i^t\}_{i,t}$ be any demands (that possibly involve user 1 both over and under-reporting). We are going to show that if user 1 changes every over-report to a truthful one, then its utility on quantum $T$ is not going to decrease. Let $T_0 \leq T$ be the last quantum where user 1 over-reported. For all users $i$ and quanta $t \in [1, T]$, let $\bar{d}_i^t = \hat{d}_i^t$, except for $\bar{d}_1^{T_0}$ which is 1's actual demand (note that $\bar{d}_1^{T_0} < \hat{d}_1^{T_0}$). Let $\bar{r}$, $\bar{R}$, $\bar{u}$, $\bar{U}$ be the result of demands $\bar{d}$. We will show that $\bar{U}_1^T \geq \hat{U}_1^T$, i.e. user 1 does not prefer the demands $\{\hat{d}_i^t\}_{i,t}$ over the demands $\{\bar{d}_i^t\}_{i,t}$. If we apply this inductively for every quantum before $T$ where user 1 over-reports, we are going to get that over-reporting is not a desirable strategy.

Up to quantum $T_0 - 1$ all users' demands in $\bar{d}$ and $\hat{d}$ are the same and thus so are the allocations: for all $i$, $\hat{R}_i^{T_0-1} = \bar{R}_i^{T_0-1}$ and $\hat{U}_i^{T_0-1} = \bar{U}_i^{T_0-1}$. Because $\hat{d}_1^{T_0} > \bar{d}_1^{T_0}$ and for $i \neq 1$, $\hat{d}_i^{T_0} = \bar{d}_i^{T_0}$, user 1 may earn some additional resources on $T_0$, i.e. $\hat{r}_1^{T_0} - \bar{r}_1^{T_0} = \hat{R}_1^{T_0} - \bar{R}_1^{T_0} = x$, for some $x \geq 0$, while other users $i \neq 1$ get less resources: $\hat{r}_i^{T_0} - \bar{r}_i^{T_0} = \hat{R}_i^{T_0} - \bar{R}_i^{T_0} \leq 0$. We first note that the $x$ additional resources that user 1 gets are in excess of 1's true demand, meaning they do not contribute towards 1's useful allocation:

$$\hat{U}_1^{T_0} - \bar{U}_1^{T_0} = \hat{R}_1^{T_0} - \bar{R}_1^{T_0} - x = 0 \tag{5.5}$$

Additionally, because user 1 does not over-report $\hat{d}$ or $\bar{d}$ in quanta $T_0 + 1$ to $T$ (by assumption $T_0$ is the last quantum before $T$ where user 1 over-reports), it holds that for $t \in [T_0+1, T]$, user 1's useful allocation is the same as the resources it receives: $\bar{u}_1^t = \bar{r}_1^t$ and $\hat{u}_1^t = \hat{r}_1^t$. This fact, combined with (5.5) proves that

$$\forall t \in [T_0, T],\ \hat{U}_1^t - \bar{U}_1^t = \hat{R}_1^t - \bar{R}_1^t - x \tag{5.6}$$

Thus, in order for this over-reporting to be a strictly better strategy, it must hold that $\hat{R}_1^T - \bar{R}_1^T > x$. We will complete the proof by proving that the opposite holds. Since there is no over-reporting in periods $t \in [T_0 + 1, T]$ we can use Lemma 3, where we use $\hat{d}_i^t \leq \bar{d}_i^t$ in place of $\hat{d}_i^t \leq d_i^t$ and summing (5.2) for all $t \in [T_0 + 1, T]$ and noticing that $\hat{d}_1^t = \bar{d}_1^t$ we get

$$\sum_k \left(\hat{R}_k^T - \bar{R}_k^T\right)^+ - \sum_k \left(\hat{R}_k^{T_0} - \bar{R}_k^{T_0}\right)^+ \leq 0$$

The above, because $(\hat{R}_k^T - \bar{R}_k^T)^+ \geq 0$, $\hat{R}_k^{T_0} - \bar{R}_k^{T_0} \leq 0$ if $k \neq 1$, and $\hat{R}_1^{T_0} - \bar{R}_1^{T_0} = x \geq 0$, proves that $\hat{R}_1^T - \bar{R}_1^T \leq x$. This completes the proof. □

Building on Lemma 1, we now prove Theorem 3.

*Proof of Online strategy proofness.* Fix a quantum $t$ and assume that all users' demands are truthful up to quantum $t - 1$. To prove that Karma is online strategy-proof we need to prove that given these conditions, user 1 cannot increase its utility on quantum $t$. Because over-reporting demand never leads to increased utility, user 1 can increase its utility only by under-reporting. However, simply under-reporting (without altering past allocations) can only decrease user 1's allocation. This completes the theorem's proof. □

While analyzing Karma properties, we encountered a new, surprising, phenomenon that may be of further theoretical interest: we show that a user that *knows all future demands of all other users* can report a demand that is lower than its actual demand in the current quantum to increase its allocation in future quanta by a small constant factor. However, any imprecision in the knowledge of all future demands of all other users could result in the user losing a factor of $\Omega(n)$ of its total allocation.

**Lemma 4.** *A user cannot increase its total useful allocation by a factor more than* 1.5× *by specifying a demand less than its real demand in any quantum. Gaining this useful allocation requires the user to know the future demands of all users. If the user does not have a precise knowledge of all future demands of all users, it can lose its useful allocation by a factor of* $\frac{n+2}{2}$ *(for* $n \geq 3$*) by specifying a demand less than its real demand.*

We provide intuition for this phenomenon using an example (Figure 5.4). In the left figure, user A is able to gain 1 extra slice in its overall allocation by under-reporting its demand (reporting 0 instead of 8) in the first quantum. By under-reporting, its allocation in the first quantum reduces, enabling it to get more resources during the second quantum when it competes with user C. In the third quantum, it is able to recover the resources it lost in the first quantum from user B, resulting in an overall gain. To see the flip-side, if the demands of other users had been as shown in Figure 5.4 (right), then user A sees a 3× degradation in overall allocation.

Before presenting the full proof of Lemma 4, we provide a brief intuitive proof sketch. To prove the first part of the Lemma 4, we consider an arbitrary user Alice and an arbitrary time period, and compare two scenarios—one where Alice is truthful (hereby called the truthful scenario) and one where Alice is deviating by under-reporting her demand during some quantum (hereby called the deviating scenario).

Our key insight for the proof is that bounding the increase in total allocation of *all users* is easier than reasoning about the increase in total allocation of an individual user (Alice) since even a small change in Alice's demand during one quantum can result in cascading effects on the total allocation of other users as well. To that end, we prove the following claim: the total amount of resources

all the users have earned in excess in the deviating scenario compared to the truthful one can be at most as large as Alice's total allocation in the truthful scenario. We prove this claim based on the following observation: whenever Alice under-reports her demand she is effectively "donating" the allocation she would have gotten in the truthful scenario to the other users whose allocations in the deviating scenario increase. Since Karma is Pareto efficient, the total gain in allocation across users during this quantum is limited by the amount donated by Alice which is in turn bound by Alice's own allocation during this quantum in the truthful scenario. By applying this reasoning iteratively across all quanta[2], we can show that the total increase in allocation across all users cannot exceed the total allocation of Alice in the truthful scenario. This already implies a $2\times$ upper bound on the maximum increase in total allocation that Alice can achieve.

To tighten the upper bound, we prove a second claim: if Alice receives higher total allocation in the deviating scenario compared to the truthful scenario, then there must exist some other user Bob who gained an even larger increase in total allocation than Alice. Putting together the above two claims allows us to establish the desired upper bound. Based on the first claim, the total gain in allocation across all users cannot exceed Alice's total allocation in the truthful scenario. This implies that the sum of total gains across Alice and Bob cannot exceed Alice's total allocation in the truthful scenario. Since Bob's gain is at least as large as Alice's gain (based on the second claim), this implies that Alice's gain is at most half the total allocation of Alice in the truthful scenario—a gain of at most $1.5\times$, thus proving the first part of Lemma 4.

[2]It turns out that Alice under-reporting in a given quantum cannot cause cascading increases in total allocation across users in future quanta if Alice does not under-report in future quanta. This is because Karma prioritizes allocation to users with high credits (or equivalently low total allocations).

The second part of the lemma is proven by first creating a set of demands where a user can under-report its demand during quantum $q$ to earn increased total allocation by some quantum $q' > q$. Then we create a set of demands that are identical up to quantum $q$ but vastly different from quanta $q+1$ to $q'$. If the user (in the hope of facing the first set of demands) under-reports its demand on quantum $q$ but ends up facing the second set of demands then this results in vastly different allocations by quantum $q'$. By correctly picking the two sets of demands we get the desired bounds.

We now present the full proof of both parts of Lemma 4.

*Proof of Lemma 4 – upper bound on potential increase.* Lemma 1 implies that it is no loss of generality to assume that user 1 does not over-report its demand, since any benefit gained by over-reporting can be gained by changing every over-report to a truthful one. This means that instead of $\hat{U}_1^t \leq 1.5U_1^t$ we can show $\hat{R}_1^t \leq 1.5R_1^t$. Towards a contradiction, let $t$ be the first quantum when user 1 gets more than 1.5 more resources by some deviation of demands, i.e. $\hat{R}_1^t > 1.5R_1^t$ and $\hat{R}_1^{t-1} \leq 1.5R_1^{t-1}$. This implies that $\hat{r}_1^t > r_1^t$, which in turn entails that there exists a user $j$ for whom $\hat{r}_j^t < r_j^t$, since the total resources allocated when user 1 is under-reporting cannot be less than those when 1 is truthful. Because $\hat{r}_1^t > r_1^t$, $\hat{r}_j^t < r_j^t$, $\hat{d}_1^t \leq d_1^t$, and $\hat{d}_j^t = d_j^t$, we can use Lemma 2 and get $\hat{R}_1^t - R_1^t \leq \hat{R}_j^t - R_j^t$. This inequality, $\hat{R}_1^t - R_1^t \geq 0$, and Lemma 3 by summing (5.2) for every quantum up to $t$, implies

$$\begin{aligned} 2\left(\hat{R}_1^t - R_1^t\right) &\leq \left(\hat{R}_1^t - R_1^t\right) + \left(\hat{R}_j^t - R_j^t\right) \\ &\leq \sum_k \left(\hat{R}_k^t - R_k^t\right)^+ \leq \sum_{\tau=1}^{t} r_1^\tau = R_1^t \end{aligned}$$

The above inequality leads to $\hat{R}_1^t \leq 1.5R_1^t$, a contradiction. □

We now prove that if users in Karma misreport their demand, they might lose a large amount of useful resources.

*Proof of Lemma 4 – decrease in useful allocation.* Consider $n$ users and $f = 1$. We will examine the first 3 quanta of two scenarios. In the first scenario, only users A, B, and C have non-zero demands, meaning we can ignore the other users. In Table 5.1 we see the demands of the users, as well as the resources they get allocated if they are truthful.

| t | 1 | 2 | 3 |
|---|---|---|---|
| A | $n$ ($\frac{n}{2}$) | $n$ ($\frac{n}{4}$) | $n$ ($\frac{3n}{8}$) |
| B | $n$ ($\frac{n}{2}$) | 0 (0) | $n$ ($\frac{5n}{8}$) |
| C | 0 (0) | $n$ ($\frac{3n}{4}$) | 0 (0) |
| others | 0 (0) | 0 (0) | 0 (0) |

Table 5.1: Example where it is in A's best interest to lie by misreporting a 0 demand on the first quantum. Each cell has the demand of the user and the parentheses contain the resources allocated if every user is truthful.

It turns out that it is in A's best interest to lie on the first quantum and report a demand of 0. This leads to A getting $\frac{5n}{4}$ resources, compared to $\frac{9n}{8}$, had A been truthful.

Now we examine the second scenario in Table 5.2, where every user other than A has the same demand.

| t | 1 | 2 | 3 |
|---|---|---|---|
| A | $n$ ($n$) | $n$ (1) | $n$ (1) |
| others | 0 (0) | 1 (1) | 1 (1) |

Table 5.2: Example where if A misreports its demand according to the demands of Table 5.1, they lose resources. Each cell has the demand of the user and the parentheses contain the resources allocated if every user is truthful.

Comparing Tables 5.1 and 5.2 we notice that A has the same demand for all quanta. This means that if A decides to misreport 0 demand on the first

quantum, in order to maximize its resources according to the first scenario, A would lose resources in the second one. More specifically, in the second scenario misreporting leads to A getting only 2 resources, compared to $n + 2$ resources for truthful reporting, i.e. misreporting is a factor of $\frac{n+2}{2}$ worse. □

All the above discussed properties and guarantees hold for the generalized algorithm where users have different fair shares (Algorithm 4). The only difference, is that the upper bound factor in Lemma 4 changes from 1.5× to 2×. We refer readers to [69] for the corresponding proofs.

Recall that Karma focuses on long-term fairness without a priori knowledge of future user demands. To that end, the following theorem summarizes Karma's fairness guarantees:

**Theorem 4.** *For any quantum $q$, given fixed user allocations from quantum 0 to quantum $q-1$, and user demands at quantum $q$, Karma maximizes the minimum total allocation from quantum 0 to quantum $q$ across users.*

The proof for the above theorem follows from the prioritized resource allocation mechanism of Karma. Given allocations from quantum 0 to $q-1$, the user with the least total allocation up to quantum $q-1$ will have the largest number of credits. In quantum $q$, Karma will prioritize the allocation of resources to this user (until it is no longer the one with the minimum total allocation, after which it will prioritize the next user with the minimum total allocation, and so on), thus maximizing the minimum total allocation from quantum 0 to $q$ across users—this is the best one can do in quantum $q$ given past allocations.

## 5.5 Integrating Karma with existing disaggregated memory systems

We have implemented Karma on top of Jiffy [118], an open-sourced elastic disaggregated memory system. Jiffy has a standard distributed data store architecture (Figure 5.5(a)): resources are partitioned into fixed-sized slices (blocks of memory) across a number of resource servers (memory servers), identified by their unique sliceIDs (referred to as blockIDs in Jiffy). A logically centralized controller tracks the available and allocated slices across the various resource servers and stores a mapping that translates sliceIDs to the corresponding resource server. We have implemented Karma as a new resource allocation algorithm at the Jiffy controller[3].

Users interact with the system through a client library that provides APIs for requesting resource allocation and accessing allocated resource slices. Users express their demands to the controller through resource requests which specify the number of slices required. The controller periodically performs resource allocation using the Karma algorithm and provides users with the sliceIDs of the resource slices that are allocated to them. Users can then directly access these slices from the resource servers through read or write API calls without requiring controller interposition. In the rest of this section, we discuss the key data structures and mechanisms required to integrate Karma with Jiffy.

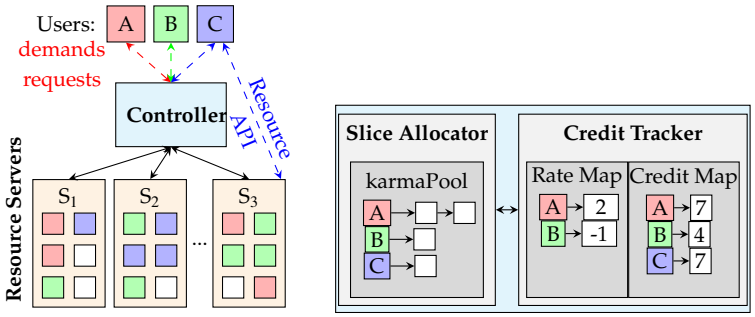
Karma employs three key data structures to efficiently implement the policies and mechanisms outlined in §5.4: karmaPool, a credit map, and a rate map.

[3]Karma can thus directly piggyback on Jiffy's existing mechanisms for controller fault tolerance [118, Section 4] to persist its state across failures.

**karmaPool.** Recall from §5.4.2 that the karmaPool tracks the pool of donated slices and shared slices, and needs to be updated when resource allocations change. Also, the resource allocation algorithm should be able to efficiently select donated slices from a particular user while satisfying borrower demands (§5.4.2). To this end, the karmaPool is implemented as a hash map, mapping userIDs to the list of sliceIDs corresponding to slices donated by them. The list of sliceIDs corresponding to shared slices is stored in a separate entry of the same hash map. When resource allocations change, the corresponding sliceIDs are added to or removed from the corresponding lists. As such, karmaPool supports all updates in $O(1)$ time.

**Credit Tracking.** Karma employs two data structures for tracking and allocating credits across various users: a rate map and a credit map. The rate map maps each user to the *rate* at which it earns or spends credits every quantum as a result of donating or borrowing slices. The rate is positive for a donor, and is equal to the number of its donated slices that are borrowed by other users; it is negative for a borrower, and is equal to the number of slices allocated to it beyond its guaranteed share. The credit map, on the other hand, maps each user to a counter corresponding to its current credits.

Separating the rate map and credit map facilitates efficient credit tracking at each quantum: Karma simply iterates through the rate map entries, and updates the credit counters in the credit map based on the corresponding user credit rates. Since the rate map only contains entries for users with non-zero rates, Karma can efficiently update credits for only the relevant users. Karma recomputes allocations—and therefore rates—only when user demands change; in quanta where demands are unchanged, advancing credits requires nothing




(a) System Architecture (b) Karma state in the Controller

Figure 5.5: Karma Design. See §5.5 for details.

more than a pass over the rate map. At the same time, employing a hash-map for each of them permits $O(1)$ updates to the user credit rate or number of credits while performing resource allocation.

**Borrowing and donating slices.** Karma realizes its credit-based prioritized allocation algorithm (§5.4.2) using two modules at the controller. First is a *slice allocator* that maintains the karmaPool to track and update slice allocations across users, and, second a *credit tracker* that maintains the current number of credits for any user (via Credit Map) and how it should be updated (via Rate Map). Figure 5.5(b) shows these modules along with the data structures they manage.

The slice allocator intercepts resource requests from users, periodically executes the Karma resource allocation algorithm (Algorithm 3) to compute allocations based on the user demands, and updates slices in the karmaPool accordingly. It interacts with the credit tracker to query and update user credits. A naïve implementation of Algorithm 3 runs in $O(n \cdot f \cdot \log n)$ time, where $n$ is the number of users, and $f$ is the fair share[4]. Instead of computing allocations one

[4]The loop in Line 10 of Algorithm 3 takes $O(n \cdot f)$ iterations. If we were to maintain min/max heaps ordered by credits for the donor and borrower sets, the donor/borrower with the minimum/maximum credits could be found in $O(1)$ time; however, each iteration changes the credits of the corresponding donor and borrower, and updating the two heaps accordingly would take $O(\log n)$ time per iteration.

slice at a time, we use an optimized implementation that carefully computes them in a batched fashion, that reduces the runtime to $O(n \log n)$. This enables the slice allocator to support resource allocation at fine-grained timescales.

Our optimized implementation builds on the observation that the loop in Line 10 of Algorithm 3 is essentially a water-filling process over user credits. Consider the case where the supply of slices exceeds the borrower demand. Here, the demands of all borrowers can be met, so the only question the loop resolves is which donors' slices are used; and, each slice drawn from a donor increases that donor's credits by exactly one. As such, the loop repeatedly draws a slice from the poorest donor until it is no longer the poorest, effectively raising the credits of the poorest donors in lockstep, one credit level at a time.

Instead of replaying this process one slice at a time, our implementation sorts donors by their credits once, and then advances the active set of donors—those currently at the lowest credit level—in batches. Specifically, it computes the largest number of slices $\Delta$ that can be drawn from each active donor before the state of the process qualitatively changes, that is, the minimum of (i) the smallest number of residual slices available for donation across active donors (beyond which that donor leaves the active set), (ii) the gap to the credit level of the next-poorest donor (beyond which new donors join the active set), and (iii) the residual borrower demand divided by the size of the active set (beyond which the loop terminates). It then advances all active donors by $\Delta$ slices at once, and assigns the last few slices individually once the residual demand drops below the size of the active set. Since each batch is delimited by an event that either adds a donor to or removes a donor from the active set (or terminates the loop), and each donor joins and leaves the active set at most once, the loop takes $O(n)$

batches rather than $O(n \cdot f)$ single-slice steps.

To make each batch cheap, we maintain the active set in a min-heap ordered by the number of residual slices available for donation, augmented with a global offset that is shared across all its elements. The offset lets us decrement the residual counts of all active donors uniformly in $O(1)$ time—the operation performed once per batch—while insert, find-min and delete-min continue to take $O(\log n)$ time.

Shared slices are modeled as a pseudo-donor with arbitrarily large credits, so that they are drawn from only after all donated slices have been exhausted. The case where borrower demand exceeds supply is handled symmetrically: all donated slices are used, slices are handed out to the richest borrowers (draining their credits in lockstep), and the same batching applies with the active set being the borrowers at the highest credit level. Overall, our implementation runs in $O(n \log n)$ time, dominated by the initial sort of users by credits, and is independent of the fair share $f$ (equivalently, the total number of slices).

**Consistent hand-off of resources.** Since users are allowed to directly access slices from resource servers, we need to ensure consistent hand-off of slices from one user to another when slices are reallocated. For example, say user $U_1$ has a slice during a given quantum, and in the next quantum, this slice is allocated to user $U_2$. We need to ensure that (1) $U_1$'s data is flushed to persistent storage before $U_2$ overwrites it (2) $U_1$ should not be able to read/write to the slice after $U_2$ has accessed it (for example, there could be in-flight read/write requests to the slice which were initiated before $U_1$ gets to know its allocation changed).

Karma ensures the above by maintaining a monotonically increasing se-

quence number and current userID for each slice, at both the controller (within the karmaPool) and the resource servers (as slice metadata). On slice allocation, its userID is updated and its sequence number is incremented at the controller, and the sequence number is returned to the user. Subsequent user reads and writes to the slice specify this userID and sequence number. A slice read succeeds only if the accompanying sequence number is the same as the current slice sequence number, while a slice write succeeds only if the accompanying sequence number is the same or greater than the current sequence number. If a write necessitates an overwrite of the current slice content and metadata, the old slice content is transparently flushed to persistent storage (*e.g.*, S3) before the overwrite. In our example above, $U_2$'s first access to the slice after re-allocation will trigger a flush of $U_1$'s data to S3 and update the slice sequence number. Following this $U_1$'s accesses to this slice will fail since the current sequence number of the slices is higher. $U_1$ can then read/write this data from persistent storage. Implementing consistent resource hand-off in Jiffy required minor changes to the controller (to track sequence numbers per slice), memory servers (to perform sequence number checking), and the client library (to tag requests with sequence numbers).

## 5.6 Evaluation

We have already established Karma properties theoretically in §5.4. In this section, we evaluate how Karma's properties translate to application-layer benefits over an Amazon EC2 testbed with real-world workloads. Our evaluation demonstrates that:

- Karma reduces the performance disparity between different users by $\sim 2.4\times$ relative to classic max-min fairness, without compromising on system-wide utilization or average performance.
- Karma incentivizes users to share resources, quantifying Karma's online strategy-proofness property.

We primarily focus on the shared cache use case from §5.3 for the following reason. While datasets for the shared data analytics clusters use case are publicly available (*e.g.*, Google and Snowflake datasets), they do not provide user queries that may impact our final conclusions. For the shared cache use case, we do have all the information we need: these datasets provide information on the working set size of each user over time, which can be fed into an end-to-end multi-tenant in-memory cache system running on Amazon EC2. We, thus, focus on this use case.

**Experimental setup.** Our experimental setup consists of a distributed elastic in-memory cache shared across multiple users backed by a remote persistent storage system. For the cache, we use Jiffy [118], augmented with our implementation of Karma (§5.5) and other evaluated schemes. If the evaluated scheme does not allocate sufficient slices to a user on Jiffy to fit its entire working set, the remaining data is accessed from remote persistent storage. When slices are reallocated between users across quanta, the corresponding data is moved between Jiffy and persistent storage through the consistent hand-off mechanism described in §5.5. We deployed our setup on Amazon EC2 using c5n.9xlarge instances (36 vCPUs, 96GB DRAM, 50Gbps network bandwidth). We host the Jiffy controller and resource servers across 7 instances and use 25 instances for the users/clients that issue queries to Jiffy. We use Amazon S3 as the persistent

storage system.

**Workload.** We use the publicly available Snowflake dataset [219] that provides dynamic user demands in terms of memory usage for each customer from Snowflake's production cluster. We use these demands as the dynamic working set size for individual users. For each user, we issue data access queries using the standard YCSB-A workload [53] (50% read, 50% write) with uniform random access distribution, with queries during each quantum being sampled (according to the YCSB parameters) within the instantaneous working set size of that user. If a query references data that is currently cached in Jiffy, then it is serviced directly from the corresponding resource server; otherwise, it is serviced from the persistent storage.

**Default parameters.** Unless specified otherwise, we randomly choose 100 users (out of $\sim$ 2000 users) over a randomly-chosen 15 minute time window (out of a 14-day period) in the Snowflake workload. To test for extreme scenarios, we set the length of each quantum to be one second (that is, a total of 900 quanta). The fair share of each user is 10 slices, and the total memory capacity of the system is set to the number of users times the fair share (1000 slices). Each slice is 128MB in size, while each query corresponds to a read or write to a 1KB chunk of data (the default size in the YCSB workload).

**Compared schemes.** We compare Karma to strict partitioning and max-min fairness, since they correspond to the two most popular fair allocation schemes, and represent extremes in resource allocation and performance. When evaluating Karma, we set the number of initial credits to a large value[5]. The fraction of

[5]As discussed in §5.4, the precise value is unimportant. Here, we set it to $900,000$, so that even if a user was allocated the full system capacity for the entire duration ($1000 \times 900$) it would

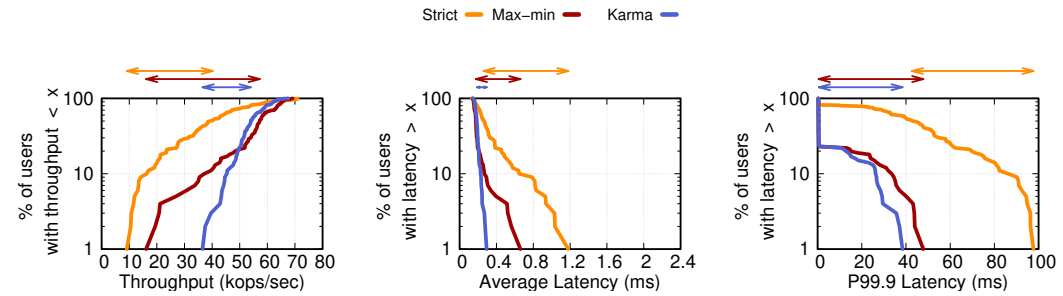
fair share that is guaranteed ($\alpha$) is 0.5 by default.

**Metrics.** We evaluate system-wide resource utilization, along with both per-user and system-wide performance—key metrics for any resource allocation mechanism. For performance, we measure both throughput and latency (average and 99.9th percentile tail). We define performance *disparity* for an allocation scheme as the ratio of median to minimum performance (that is, throughput or latency) observed across various users. For any given user, we define *welfare* over time $t$ as $\frac{\sum_t \text{allocations}}{\sum_t \text{demands}}$, that is, the fraction of its total demands satisfied by the allocation scheme. We define *fairness* as $\frac{min_{users}\text{welfare}}{max_{users}\text{welfare}}$ (higher is better, 1 is optimal), as a measure of welfare disparity between users.

We now evaluate Karma's benefits in terms of reducing disparity across users' application-level performance as well as resource allocation.

**Karma reduces performance disparity between users.** Figure 5.6(a) shows the throughput distribution across users for our compared schemes; the y-axis is presented in log-scale to focus on the users at the tail of the distribution, which observe the most performance disparity. Since Karma strives to balance fairness over time, it significantly narrows the throughput distribution across users compared to the two baselines: the ratio between the maximum and minimum throughput across all users is 7.8× with strict partitioning and 4.3× with max-min fairness, but only 1.8× for Karma. As Figure 5.6(d) shows, Karma lowers the throughput disparity across users by 2.4× compared to max-min fairness. Karma also reduces average latency disparity (Figure 5.6(b)) by 2.4× and 99.9th percentile latency disparity (Figure 5.6(c)) by 1.2× compared to max-min fairness by enabling a tighter distribution for both latencies.

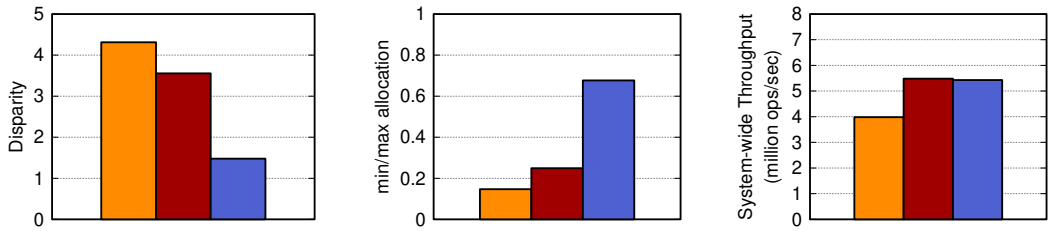
not run out of credits.




(a) CDF of throughput across users

(b) CCDF of average latency across users

(c) CCDF of P99.9 latency across users




(d) Disparity in throughput across users (median/min)

(e) Fairness in overall allocations (min/max allocation)

(f) System-wide average throughput (million ops/sec)

Figure 5.6: Understanding Karma benefits. (a) Karma enables a much tighter throughput distribution across users (colored arrows show the absolute gap between median and minimum throughput across users). (b, c) It also enables a tighter distribution of average and tail latencies across users (again, colored arrows show the absolute gap between median and maximum latency across users). (d) Karma achieves much lower throughput disparity—ratio of median to minimum values of throughput across users—than classic max-min fairness. (e) It also significantly reduces the gap between the users with minimum and maximum overall allocations, (f) while achieving similar system-wide performance as max-min fairness.

Equitability in performance across users for a scheme is closely tied to how fairly resources are allocated across users. Specifically, because of the large gap between elastic memory (Jiffy) and S3 latencies (50–100×), accesses to slices in S3 result in significantly lower throughput than accesses to slices in elastic memory. As a result, users' average throughput ends up being roughly proportional to their total allocation of slices in elastic memory over time. Similarly, since a larger total allocation results in a smaller fraction of requests going to S3, average and tail latencies also reduce.

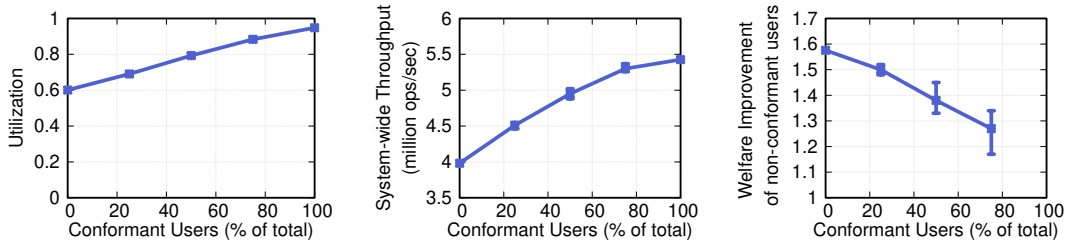
**Karma reduces disparity in allocations.** We now quantify disparities in overall allocations obtained by users across our compared schemes via our fairness metric in Figure 5.6(e). Due to dynamic demands, strict partitioning exhibits very poor fairness, since users with very bursty demands end up getting much lower total allocations than users who have steady demands[6]. While max-min fairness observes better fairness compared to strict partitioning, the best-off user still receives 4× higher allocation than the worst-off user, resulting in poor absolute fairness. Karma achieves significantly better fairness with the best-off user receiving only 1.5× higher allocation than the worst-off user. It is able to achieve this by prioritizing the allocation of resources beyond the fair share to users with more credits (§5.4.2).

**Karma achieves Pareto efficiency and high system-wide performance.** Karma achieves the same overall resource utilization as max-min fairness ($\sim 95\%$). This is because Karma is Pareto efficient (§5.4.3) similar to max-min fairness and thus achieves near-optimal utilization. We find that the optimal utilization is $< 100\%$ since some quanta observe total user demands less than system capacity.

Max-min fairness observes 1.4× higher system-wide throughput (that is, throughput aggregated across all users) than strict partitioning (Figure 5.6(f)) since it permits allocations beyond the fair share, allowing more requests to be served on faster elastic memory. Karma observes system-wide performance similar to max-min fairness for similar reasons; the slight variations are attributed to variance in S3 latencies.

We now empirically demonstrate that Karma incentivizes users to donate re-

[6]Note that only *useful* allocations are considered—strict partitioning guarantees a fixed allocation at all times, but resources may remain unused when demand is low.



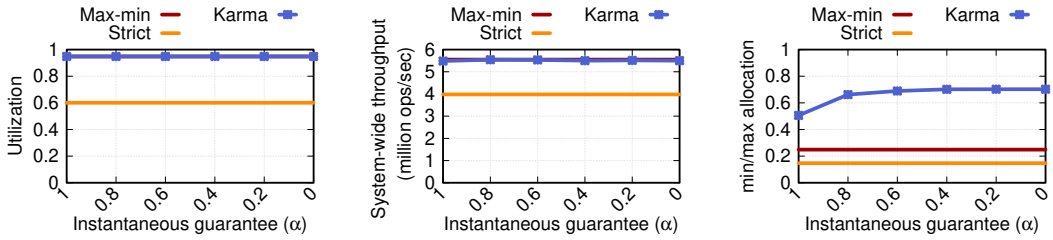

(a) Utilization (b) Performance (c) Welfare improvement

Figure 5.7: Karma incentivizes resource sharing. All metrics are computed as averages (with error bars) for three random selections of users being non-conformant.

sources instead of hoarding them, to improve their own as well as overall system welfare. To this end, we vary the fraction of users using Karma that are *conformant* or *non-conformant*. A conformant user is truthful about its demands and donates its resources when its demand is less than its fair share. A non-conformant user, on the other hand, always asks for the maximum of its demand or its fair share (that is, it over-reports its demand during some quanta).

**Resource utilization and system-wide performance improve with more conformant users.** Figure 5.7(a) and Figure 5.7(b) show that Karma's system-wide utilization and performance improve as the fraction of conformant users increases. This is because as more users donate resources when they do not need them, other users can use these resources, improving overall utilization and performance. When none of the users are conformant, since no one ever donates any resources, Karma essentially reduces to strict partitioning, hence achieving low overall utilization and performance. When all users are conformant, Karma achieves optimal utilization and performance, similar to classic max-min fairness.

**Becoming conformant improves user welfare.** Figure 5.7(c) shows the average welfare gain non-conformant users would achieve if they were to become




(a) Utilization (b) Performance (c) Fairness

Figure 5.8: Sensitivity analysis with varying instantaneous guarantee ($\alpha$). (a, b) Karma matches the resource utilization and system-wide performance of max-min fairness independent of $\alpha$ (c) Smaller values of $\alpha$ result in improved long-term fairness.

conformant. When non-conformant users become conformant, it leads to significant (1.17–1.6×) welfare gains for them, empirically validating Karma's property that users have nothing to gain by over-reporting their demand (§5.4.3). Note that the gain varies with the number of conformant users in the system—the gains from non-conformant users becoming conformant are higher when the percentage of conformant users is low. As expected, the gains show diminishing returns as more users in the system become conformant as overall utilization is already high.

We now show sensitivity analysis with the only parameter in the Karma algorithm—the instantaneous guarantee ($\alpha$). Figure 5.8 shows he results with varying $\alpha$. Karma continues to match the resource utilization and system-wide performance of max-min fairness independent of $\alpha$ (Figure 5.8(a) and Figure 5.8(b)). Varying $\alpha$ has an impact on the long-term fairness achieved by Karma (Figure 5.8(c)), with smaller values of $\alpha$ resulting in improved fairness, thus validating our discussion in §5.4. Even for $\alpha = 1$, Karma is able to achieve significantly better fairness compared to max-min fairness. This is because, while it allocates resources up to the fair share identically to max-min fairness, it prioritizes allocation beyond the fair share based on credits.

## 5.7 Related work

There is a large and active body of work on resource allocation and scheduling, exploring various models and settings; it would be a futile attempt to compare Karma with each individual work. We do not know of any other resource allocation mechanism that guarantees Pareto efficiency, strategy-proofness, and fairness similar to Karma for the case of dynamic user demands; nevertheless, we discuss below the most closely related works.

**Max-min fairness variants in cloud resource allocation and cluster scheduling.** Many works study variants of max-min fairness for cloud resource allocation and cluster scheduling [16, 23, 48, 78, 82–84, 140, 145, 179–181, 195, 198, 231], including recent work on ML job scheduling [42, 85, 146, 165, 176]. We make three important notes here. First, while dominant resource fairness (DRF) [78] has generalized max-min fairness to multiple resources, it makes the same assumptions as max-min fairness: user demands being static over time; our goals are different: we have identified and resolved the problems with max-min fairness for the case of a single resource but over dynamic user demands. It is an interesting open problem to generalize Karma for the case of multiple resources.

Second, cluster scheduling has been studied under several metrics beyond fair resource allocation (*e.g.*, job completion time, data locality, priorities, etc.). Themis [146] considers long-term fairness but defines a new ML workload-specific notion of fairness, and is therefore not directly comparable to Karma. Our goals are most aligned with those works that study fair allocation under strategic users while guaranteeing Pareto efficiency. To that end, the closest to Karma is CARBYNE [83]. However, CARBYNE not only assumes non-strategic

users but also, for the single-resource case (the focus of this chapter), CARBYNE converges to max-min fairness. As discussed earlier, generalizing Karma to multiple resources remains an open problem; a solution for that problem must be compared against CARBYNE.

Finally, fairness in application-perceived performance is only indirectly related to fairness in resource allocation: other factors like software systems (*e.g.*, hypervisors and storage systems) and resource preemption granularity can impact performance. Similar to other mechanisms [20, 23, 44, 78, 82–84, 106, 144, 145,181,188,198,201,219,231,238], Karma's properties are independent of these system-level factors; while our evaluation shows that Karma properties translate to application-level benefits, absolute numbers depend on the underlying system implementation.

**Allocation of time-shared resources.** Generalized Processor Sharing (GPS) [174] is an idealized algorithm for sharing a network link which assumes that traffic is infinitesimally divisible (fluid model). For equal-sized packets and equal flow weights, GPS reduces to Uniform Processor Sharing [174, Section 2], which is equivalent to max-min fairness. GPS guarantees fairness over arbitrary time intervals only under the assumption that flows are *continuously backlogged* [174, Section 2]. This assumption implies that flows always have demand greater than their fair share, making it trivial to guarantee a max-min fair share of the network bandwidth over arbitrary time intervals. Classical fair-queueing algorithms [28,60,155,197,241] in computer networks approximate GPS with the constraint of packet-by-packet scheduling. Under this constraint, varying-sized packets and different flow weights make it hard to realize fairness efficiently; thus, the technical question that these algorithms solve is to achieve fairness

approximately equal to GPS with minimal complexity. Karma focuses on a different problem—we show that GPS guarantees (equivalent to max-min fairness) are not sufficient when demands are dynamic and present new mechanisms to achieve fairness while maintaining other properties for such dynamic demands.

Stride [221] scheduling essentially approximates GPS in the context of CPU scheduling [221, Section 7], and thus the above discussion applies to it as well. DRF-Q [77] generalizes DRF to support both space and time-shared resources, but is explicitly designed to be memoryless similar to max-min fairness, and therefore suffers from similar issues for long-term fairness. Least Attained Service (LAS) [31, 124, 171] is a classical job scheduling algorithm that has been applied to packet scheduling [31], GPU cluster scheduling [85], and memory controller scheduling [122]. For $\alpha = 0$, Karma behaves similarly to LAS, and for $\alpha > 0$, Karma generalizes LAS with instantaneous guarantees. Moreover, our results from §5.4.3 establish strategy-proofness properties of LAS for dynamic user demands, which may be of independent interest.

**Theory works.** Several recent papers in the theory community study the problem of resource allocation for dynamic user demands. Freeman et al. [72] and Hossain et al. [92] consider dynamic demands under a different setting, where users can benefit when they are allocated resources above their demand; under this setting, they focus on instantaneous fairness (which is non-trivial since users can be allocated resources beyond their demand). Karma instead focuses on long-term fairness under the traditional model, where users do not benefit from resources beyond their demands. Sadok et al. [191] present minor improvements over max-min fairness for dynamic demands. Their mechanism allocates resources in a strategy-proof manner according to max-min fairness

while marginally penalizing users with larger past allocations using a parameter $\delta \in [0, 1)$. For both $\delta = 0$ and $\delta \to 1$, the penalty goes to 0 for every past allocation, and the mechanism becomes identical to max-min fairness; for other values of $\delta$, the penalty is at most a $\delta(1 - \delta) \leq 1/4$ fraction of past allocation surplus, and it reduces exponentially with time (users who were allocated large amounts of resources further in the past receive an even smaller penalty). Thus, for all values of $\delta$, and in particular, for $\delta = 0$ and $\delta \to 1$, their mechanism suffers from the same problems as max-min fairness. Aleksandrov et al. [8] and Zeng et al. [240] consider dynamic demands, but in a significantly different setting than ours where resources arrive over time. Finally, Fikioris et al. [69] is a follow-up work that extends Karma theoretical analysis to the scenarios where users have different fair shares, multiple users collude with each other and more than one resource type is allocated.

**Pricing- and credit-based resource allocation.** Another stream of work related to Karma is pricing-based and bidding-based mechanisms for resource allocation, *e.g.*, spot instance marketplace and virtual machine auctions [5, 12, 73, 229, 244, 245]. While interesting, this line of work does not focus on fair resource allocation and is not applicable to use cases that Karma targets. XChange [227] proposes a market-based approach to fair resource allocation in multi-core architectures but focuses on instantaneous fairness rather than long-term fairness, unlike Karma. It assigns a "budget" of virtual currency to each user which can be used to bid for resources. This budget is however reset during every time quantum, and therefore information about past allocations is not carried over.

Credits are used in many other game theoretic contexts [68, 170, 189], *e.g.*, in peer-to-peer and cooperative caching settings to incentivize good behavior

among participants with static demands [55, 178, 232]. However, we are not aware of any credit-based mechanisms that deal with resource allocation in the context of dynamic user demands.

## 5.8 Broader implications

In this chapter, we considered the problem of allocating disaggregated memory resources across multiple users. We demonstrated that the classical max-min fairness algorithm for resource allocation loses one or more of its desirable properties—Pareto efficiency, strategy-proofness, and/or fairness—for the realistic case of dynamic user demands. We presented Karma, a resource allocation mechanism for dynamic user demands, and theoretically established Karma guarantees related to Pareto efficiency, strategy-proofness, and fairness for dynamic user demands. Experimental evaluation of a realization of Karma in a multi-tenant disaggregated memory system demonstrated that Karma's theoretical properties translate well into practice: it reduces application-level performance disparity by as much as 2.4× when compared to max-min fairness while maintaining high resource utilization and system-wide performance.

Beyond disaggregated memory resources, Karma's core mechanisms apply more broadly to the allocation of any resource of a single type (*e.g.*, compute, network bandwidth, GPUs). Indeed, applying Karma to other use cases is an interesting direction for future work and may require additional extensions (*e.g.*, handling all-or-nothing or gang-scheduling constraints which are prevalent in the context of GPU resource allocation [42, 146]).

# CHAPTER 6

# CONCLUSION AND FUTURE DIRECTIONS

This dissertation advances the status-quo of disaggregated memory management through four key contributions. In Chapter 2, we built an in-depth understanding of the host interconnects that facilitate memory accesses. In Chapter 3, we extended our understanding to include cache-coherent interconnects for memory disaggregation (*e.g.*, CXL), and characterized the impact of data placement across local and disaggregated memory on application performance. In Chapter 4, we presented Colloid, an operating system memory management mechanism that builds on our understanding of host interconnects to adapt data placement in a way that maximizes the performance of applications running atop disaggregated memory architectures. In Chapter 5, we presented Karma, a mechanism for allocating disaggregated memory resources across tenants under dynamic demands.

§1.3 provided a discussion of the broader technical challenges that need to be addressed in order to realize the ideal vision of memory disaggregation. We now discuss specific directions of future work motivated by this dissertation both in the context of memory disaggregation and more generally.

**Memory management mechanisms beyond Colloid.** Our discussion of Colloid in Chapter 4 focuses on minimizing average access latency. However, the end-to-end latency of memory requests also includes address translation latency—the time the processor's MMU takes to translate from virtual to physical addresses. Address translation latency depends on the choice of page size for different regions of the application's virtual address space. Using hugepages minimizes the average address translation latency due to improved TLB lo-

cality; however, it may lead to sub-optimal average access latency, since the hottest data cannot always be placed in lower-latency memory. On the other hand, using pages presents the opposite trade-off. Neither extreme of using all (huge)pages is guaranteed to maximize application performance in the general case. To that end, it would be interesting to extend operating system memory management mechanisms to jointly optimize page size determination and page placement algorithms to minimize the average sum of translation and access latencies. Furthermore, as briefly illustrated in Chapter 4, the placement of not just data pages, but also page table pages, impacts application performance. To that end, designing memory management mechanisms that adapt the placement of both data and page table pages is an interesting future direction.

**Memory bandwidth management.** Traditionally, OS memory management has focused on virtualizing memory capacity. As discussed in Chapters 1 and 2, memory bandwidth is increasingly becoming a prominent bottleneck. Our observations in Chapter 2 show that the memory bandwidth allocations received by co-located applications are an artifact of workloads characteristics and low-level hardware details (*e.g.*, credits, and latencies of host interconnects). This makes it difficult to provide applications predictable memory bandwidth allocations and more generally host network resources. It would be interesting to explore abstractions, mechanisms and hardware interfaces to enable operating system controlled virtualization of memory bandwidth (for both local and disaggregated memory). Building on such mechanisms, it would also be interesting to apply an algorithm like Karma (discussed in Chapter 5) to (re)allocate memory bandwidth across applications over time.

**Managing memory distributed across coherence domains.** As discussed in

§1.3, memory disaggregation over interconnects like CXL does not necessarily guarantee hardware cache coherence across all hosts. This results in multiple coherence domains each containing one or more hosts. In fact, even all the memory within a single modern host is not guaranteed to be hardware cache-coherent. For example, GPUs are often connected to CPUs over the traditional peripheral interconnect (PCIe) which is not cache-coherent. As a result, the CPU memory and the GPU memory form two separate coherence domains. Another example where memory is distributed across coherence domains is in wafer-scale chips which contain hundreds of thousands of processors each with their own independent memories that are not hardware cache-coherent. Managing memory distributed across coherence domains raises basic questions regarding the virtual memory abstraction provided by the operating system. For example, the operating system could continue to provide applications with the abstraction of a single virtual address space, in which case it would have to implement the necessary synchronization mechanisms at the software-level. Alternatively, it could expose multiple address spaces to the application/compiler. It would be interesting to explore this design space while drawing upon insights from classical distributed shared memory literature and potentially revisiting classical trade-offs given the properties of host interconnects.

**The host as a distributed system.** In Chapters 2 and 3, we demonstrated that the performance of data transfers with the host is governed by the interplay between the processor, peripheral and memory interconnects. This suggests that these interconnects should collectively be viewed as a *network* interconnecting the various devices within a host. Furthermore, as discussed above, memory in modern hosts is often distributed across multiple coherence domains. Finally, in modern hosts, any individual compute, memory or network device can fail

independently without necessarily rendering the entire host unavailable. Consequently, it may be unnecessarily expensive and/or inefficient for the OS to continue abstracting the host as a unit of failure. All the above observations put together suggest that it may be better to model each host itself as a distributed system (revisiting an old idea [24, 25])—that is, a collection of independent devices connected over a host network. We believe it would be very interesting and timely to rethink operating system and distributed system design and principles through this lens. By providing a framework to understand the properties and performance of the host network, the contributions of this dissertation lay the foundation for such an endeavor (as the properties and performance of a distributed system often depend on the minimal set of assumptions that can be made about the underlying network).

## BIBLIOGRAPHY


[1] Firas Abuzaid, Srikanth Kandula, Behnaz Arzani, Ishai Menache, Matei Zaharia, and Peter Bailis. Contracting Wide-Area Network Topologies to Solve Flow Problems Quickly. In *USENIX NSDI*, 2021.

[2] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-Transparent Page Management for Two-Tiered Main Memory. In *ACM ASPLOS*, 2017.

[3] Saksham Agarwal, Rachit Agarwal, Behnam Montazeri, Masoud Moshref, Khaled Elmeleegy, Luigi Rizzo, Marc Asher de Kruijf, Gautam Kumar, Sylvia Ratnasamy, David Culler, and Amin Vahdat. Understanding Host Interconnect Congestion. In *ACM HotNets*, 2022.

[4] Saksham Agarwal, Arvind Krishnamurthy, and Rachit Agarwal. Host Congestion Control. In *ACM SIGCOMM*, 2023.

[5] Orna Agmon Ben-Yehuda, Eyal Posener, Muli Ben-Yehuda, Assaf Schuster, and Ahuva Mu'alem. Ginseng: Market-Driven Memory Allocation. In *ACM VEE*, 2014.

[6] Jung Ho Ahn, Mattan Erez, and William J Dally. The Design Space of Data-Parallel Memory Systems. In *IEEE/ACM SC*, 2006.

[7] Hasan Al Maruf and Mosharaf Chowdhury. Effectively Prefetching Remote Memory with Leap. In *USENIX ATC*, 2020.

[8] Martin Aleksandrov and Toby Walsh. Strategy-Proofness, Envy-Freeness and Pareto Efficiency in Online Fair Division with Additive Utilities. In *PRICAI*, 2019.

[9] Mohammad Alian, Siddharth Agarwal, Jongmin Shin, Neel Patel, Yifan Yuan, Daehoon Kim, Ren Wang, and Nam Sung Kim. IDIO: Network-Driven, Inbound Network Data Orchestration on Server Processors. In *IEEE/ACM MICRO*, 2022.

[10] Anthony Alles. ATM Internetworking. In *Engineering InterOp*, 1995.

[11] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can Far Memory Improve Job Throughput? In *ACM EuroSys*, 2020.

[12] Amazon. Amazon EC2 Spot Instances. https://aws.amazon.com/ec2/spot/.

[13] AMD. HyperTransport Technology: Simplifying System Design. https://www.all-electronics.de/files/2025/10/21/12912798390.pdf, 2002.

[14] AMD. Performance Monitor Counters for AMD Family 1Ah Model 00h0Fh Processors. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/programmer-references/58550-0.01.pdf, 2024.

[15] AMD. Engineering the Future of AI: How AMD Interconnects, Infinity Fabric, and Advanced Packaging Drive Scalable Compute. https://www.amd.com/en/blogs/2025/engineering-the-future-of-ai.html, 2025.

[16] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. End-to-end Performance Isolation Through Virtual Datacenters. In *USENIX OSDI*, 2014.

[17] Sotiris Apostolakis, Chris Kennelly, Xinliang David Li, and Parthasarathy Ranganathan. Necro-Reaper: Pruning Away Dead Memory Traffic in Warehouse-Scale Computers. In *ACM ASPLOS*, 2025.

[18] JEDEC Solid State Technology Association. JESD79-4: DDR4 SDRAM Standard. https://www.jedec.org/standards-documents/docs/jesd79-4a, 2021.

[19] JEDEC Solid State Technology Association. JESD79-5C: DDR5 SDRAM Standard. https://www.jedec.org/standards-documents/docs/jesd79-5c, 2024.

[20] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *ACM SIGMETRICS*, 2012.

[21] Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H Loh, and Onur Mutlu. Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems. In *IEEE/ACM ISCA*, 2012.

[22] Jens Axboe. axboe/fio: Flexible I/O Tester. `https://github.com/axboe/fio`, 2024.

[23] Hitesh Ballani, Keon Jang, Thomas Karagiannis, Changhoon Kim, Dinan Gunawardena, and Greg O'Shea. Chatty Tenants and the Cloud Network Sharing Problem. In *USENIX NSDI*, 2013.

[24] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *ACM SOSP*, 2009.

[25] Andrew Baumann, Simon Peter, Adrian Schüpbach, Akhilesh Singhania, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. Your Computer Is Already a Distributed System. Why Isn't Your OS? In *ACM HotOS*, 2009.

[26] Scott Beamer. GAPBS PageRank Implementation. `https://github.com/sbeamer/gapbs/blob/master/src/pr.cc`, 2023.

[27] Scott Beamer, Krste Asanovic, and David A. Patterson. The GAP Benchmark Suite. `http://arxiv.org/abs/1508.03619`, 2015.

[28] Jon CR Bennett and Hui Zhang. Hierarchical Packet Fair Queueing Algorithms. In *IEEE/ACM TON*, 1997.

[29] Benjamin Berg, Daniel S Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory Ganger. The CacheLib Caching Engine: Design and Experiences at Scale. In *USENIX NSDI*, 2020.

[30] Daniel S. Berger, Daniel Ernst, Huaicheng Li, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Lisa Hsu, Ishwar Agarwal, Mark D. Hill, and Ricardo Bianchini. Design Tradeoffs in CXL-Based Memory Pools for Public Cloud Platforms. In *IEEE Micro*, 2023.

[31] Ernst W Biersack, Bianca Schroeder, and Guillaume Urvoy-Keller. Scheduling in Practice. In *ACM SIGMETRICS PER*, 2007.

[32] William Bolosky, Robert Fitzgerald, and Michael Scott. Simple but Effective Techniques for NUMA Memory Management. In *ACM SOSP*, 1989.

[33] Timothy Brecht. On the Importance of Parallel Application Placement in NUMA Multiprocessors. In *USENIX SEDMS*, 1993.

[34] Qizhe Cai, Mina Tahmasbi Arashloo, and Rachit Agarwal. dcPIM: Near-Optimal Proactive Datacenter Transport. In *ACM SIGCOMM*, 2022.

[35] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding Host Network Stack Overheads. In *ACM SIGCOMM*, 2021.

[36] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. Towards $\mu$s Tail Latency and Terabit Ethernet: Disaggregating the Host Network Stack. In *ACM SIGCOMM*, 2022.

[37] Amanda Carbonari and Ivan Beschasnikh. Tolerating Faults in Disaggregated Datacenters. In *ACM HotNets*, 2017.

[38] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and Batch Processing in a Single Engine. In *IEEE DEBull*, 2015.

[39] Justin Castilla. Clustering in Redis. `https://developer.redis.com/operate/redis-at-scale/scalability/lustering-in-redis/`, 2024.

[40] Kevin K. Chang. Understanding and Improving the Latency of DRAM-Based Memory Systems. `https://kilthub.cmu.edu/articles/thesis/Understanding_and_Improving_the_Latency_of_DRAM-Based_Memory_Systems/6724127`, 2017.

[41] Kevin K Chang, Abhijith Kashyap, Hasan Hassan, Saugata Ghose, Kevin Hsieh, Donghyuk Lee, Tianshi Li, Gennady Pekhimenko, Samira Khan, and Onur Mutlu. Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization. In *ACM SIGMETRICS*, 2016.

[42] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing Efficiency and Fairness in Heterogeneous GPU Clusters for Deep Learning. In *ACM EuroSys*, 2020.

[43] Shuang Chen, Christina Delimitrou, and José F Martínez. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In *ACM ASPLOS*, 2019.

[44] Yue Cheng, Ali Anwar, and Xuejing Duan. Analyzing Alibaba's Co-Located Datacenter Workloads. In *IEEE BigData*, 2018.

[45] Albert Cho, Anish Saxena, Moinuddin Qureshi, and Alexandros Daglis. COAXIAL: A CXL-Centric Memory System for Scalable Servers. In *IEEE/ACM SC*, 2024.

[46] Hyojin Choi, Jongbok Lee, and Wonyong Sung. Memory Access Pattern-Aware DRAM Performance Model for Multi-Core Systems. In *IEEE ISPASS*, 2011.

[47] Chiachen Chou, Aamer Jaleel, and Moinuddin Qureshi. BATMAN: Techniques for Maximizing System Bandwidth of Memory Systems with Stacked-DRAM. In *ACM MEMSYS*, 2017.

[48] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. HUG: Multi-Resource Fairness for Correlated and Elastic Demands. In *USENIX NSDI*, 2016.

[49] Robert Cole, David Shur, and Curtis Villamizar. IP over ATM: A Framework Document. `https://datatracker.ietf.org/doc/html/rfc1932`, 1996.

[50] CCIX Consortium. An Introduction to CCIX. `https://www.ccixconsortium.com/wp-content/uploads/2019/11/CCIX-White-Paper-Rev111219.pdf`, 2019.

[51] CXL Consortium. Compute Express Link (CXL) Specification, Revision 2.0. `https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-2.0-Specification.pdf`, 2020.

[52] Gen-Z Consortium. Gen-Z: High-Performance Interconnect for the Data-Centric Future. `https://www.opencompute.org/files/OCP-GenZ-March-2018-final.pdf`, 2018.

[53] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *ACM SoCC*, 2010.

[54] Thomas M. Coughlin and Jim Handy. Higher Performance and Capacity with OMI Near Memory. In *IEEE HOTI*, 2021.

[55] Landon P Cox and Brian D Noble. Samsara: Honor Among Thieves in Peer-to-Peer Storage. In *ACM OSR*, 2003.

[56] Alexander D'Amour, Hansa Srinivasan, James Atwood, Pallavi Baljekar, D Sculley, and Yoni Halpern. Fairness Is Not Static: Deeper Understanding of Long Term Fairness via Simulation Studies. In *ACM FAccT*, 2020.

[57] Debendra Das Sharma, Robert Blankenship, and Daniel Berger. An Introduction to the Compute Express Link (CXL) Interconnect. In *ACM CSUR*, 2024.

[58] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *ACM ASPLOS*, 2013.

[59] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *USENIX OSDI*, 2008.

[60] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *ACM SIGCOMM*, 1989.

[61] Travis Downs. It's Not Write Combining. `https://github.com/Kobzol/hardware-effects/issues/1`, 2018.

[62] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *USENIX NSDI*, 2014.

[63] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. Towards an Adaptable Systems Architecture for Memory Tiering at Warehouse-Scale. In *ACM ASPLOS*, 2023.

[64] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N Patt. Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multicore Memory Systems. In *ACM TOCS*, 2010.

[65] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. SAP HANA Database: Data Management for Modern Business Applications. In *ACM SIGMOD Record*, 2012.

[66] Alireza Farshin, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. Make the Most Out of Last Level Cache in Intel Processors. In *ACM EuroSys*, 2019.

[67] Alireza Farshin, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostic. Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-Hundred-Gigabit Networks. In *USENIX ATC*, 2020.

[68] Joan Feigenbaum and Scott Shenker. Distributed Algorithmic Mechanism Design: Recent Results and Future Directions. In *Current Trends in Theoretical Computer Science: The Challenge of the New Century Vol 1: Algorithms and Complexity Vol 2: Formal Models and Semantics*, 2004.

[69] Giannis Fikioris, Rachit Agarwal, and Éva Tardos. Incentives in Dominant Resource Fair Allocation Under Dynamic Demands. `https://arxiv.org/abs/2109.12401`, 2021.

[70] Cache Forge. memcached - A Distributed Memory Object Caching System. `https://memcached.org/`, 2024.

[71] Henry J Fowler. TMN-Based Broadband ATM Network Management. In *IEEE Communications Magazine*, 1995.

[72] Rupert Freeman, Seyed Majid Zahedi, Vincent Conitzer, and Benjamin C. Lee. Dynamic Proportional Sharing: A Game-Theoretic Approach. In *ACM SIGMETRICS*, 2018.

[73] Liran Funaro, Orna Agmon Ben-Yehuda, and Assaf Schuster. Ginseng: Market-Driven LLC Allocation. In *USENIX ATC*, 2016.

[74] Alex Galis, Dieter Gantenbein, Stefan Covaci, Carlo Bianza, Fotis Karayannis, and George Mykoniatis. Toward Multidomain Integrated Network Management for ATM and SDH Networks. In *Broadband Strategies and Technologies for Wide Area and Local Access Networks*, 1996.

[75] Peter X Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. pHost: Distributed Near-Optimal Datacenter Transport over Commodity Network Fabric. In *ACM CoNEXT*, 2015.

[76] Yingqiang Ge, Shuchang Liu, Ruoyuan Gao, Yikun Xian, Yunqi Li, Xiangyu Zhao, Changhua Pei, Fei Sun, Junfeng Ge, Wenwu Ou, and Yongfeng Zhang. Towards Long-Term Fairness in Recommendation. In *ACM WSDM*, 2021.

[77] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. Multi-Resource Fair Queueing for Packet Processing. In *ACM SIGCOMM*, 2012.

[78] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *USENIX NSDI*, 2011.

[79] Neha Gholkar, Jovan Stojkovic, Hasan Al Maruf, Gregory Price, Prakash Chauhan, Hiral Patel, Cedric Van Goethem, Kiran Vemuri, Kishore Sriadibhatla, Kalyan Subramanian, Shobhit Kanaujia, Chunqiang Tang, and Abhishek Dhanotia. Vistara: Making CXL Real—Full Path from ASIC Design and OS Support to Hyperscale Deployment. In *IEEE/ACM ISCA*, 2026.

[80] Saugata Ghose, Hyodong Lee, and José F Martínez. Improving Memory Scheduling via Processor-Side Load Criticality Information. In *IEEE/ACM ISCA*, 2013.

[81] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In *USENIX OSDI*, 2014.

[82] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-Resource Packing for Cluster Schedulers. In *ACM SIGCOMM*, 2014.

[83] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic Scheduling in Multi-Resource Clusters. In *USENIX OSDI*, 2016.

[84] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. GRAPHENE: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters. In *USENIX OSDI*, 2016.

[85] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *USENIX NSDI*, 2019.

[86] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient Memory Disaggregation with Infiniswap. In *USENIX NSDI*, 2017.

[87] Nagendra Gulur, Mahesh Mehendale, R Manikantan, and R Govindarajan. Bi-Modal DRAM Cache: Improving Hit Rate, Hit Latency and Bandwidth. In *IEEE/ACM MICRO*, 2014.

[88] Nagendra Gulur, Mahesh Mehendale, Raman Manikantan, and Ramaswamy Govindarajan. Anatomy: An Analytical Model of Memory System Performance. In *ACM SIGMETRICS PER*, 2014.

[89] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. Re-Architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *ACM SIGCOMM*, 2017.

[90] Andrew Herdrich, Marcel David Cornu, and Khawar Munir Abbasi. Introduction to Memory Bandwidth Allocation. `https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-memory-bandwidth-allocation.html`, 2019.

[91] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving High Utilization with Software-Driven WAN. In *ACM SIGCOMM*, 2013.

[92] Ridi Hossain. Sharing Is Caring: Dynamic Mechanism for Shared Resource Ownership. In *AAMAS*, 2019.

[93] Shuihai Hu, Wei Bai, Gaoxiong Zeng, Zilong Wang, Baochen Qiao, Kai Chen, Kun Tan, and Yi Wang. Aeolus: A Building Block for Proactive Transport in Datacenters. In *ACM SIGCOMM*, 2020.

[94] Yibo Huang, Haowei Chen, Newton Ni, Yan Sun, Vijay Chidambaram, Dixin Tang, and Emmett Witchel. Tigon: A Distributed Database for a CXL Pod. In *USENIX OSDI*, 2025.

[95] Ying Huang. [PATCH -V4 0/3] Memory Tiering: Hot Page Selection. https://lwn.net/ml/linux-kernel/20220622083519.708236-1-ying.huang@intel.com/, 2022.

[96] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. Rearchitecting Linux Storage Stack for $\mu$s Latency and High Throughput. In *USENIX OSDI*, 2021.

[97] Intel. An Introduction to the Intel QuickPath Interconnect. https://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html, 2009.

[98] Intel. Intel Data Direct I/O Technology (Intel DDIO): A Primer. https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf, 2012.

[99] Intel. Intel Xeon Processor Scalable Family Technical Overview. https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html, 2017.

[100] Intel. Intel Xeon Processor Scalable Memory Family Uncore Performance Monitoring. https://kib.kiev.ua/x86docs/Intel/PerfMon/336274-001.pdf, 2017.

[101] Intel. 3rd Gen Intel Xeon Processor Scalable Family, Codename Ice Lake, Uncore Performance Monitoring. https://cdrdv2-public.intel.com/679093/639778%20ICX%20UPG%20v1.pdf, 2021.

[102] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual. https://cdrdv2.intel.com/v1/dl/getContent/671200, 2023.

[103] Intel. Intel Xeon CPU Max Series. https://www.intel.com/content/www/us/en/products/details/processors/xeon/max-series.html, 2024.

[104] Intel. Sapphire Rapids (SPR) Uncore Events. https://github.com/intel/perfmon/blob/main/SPR/events/sapphirerapids_uncore_experimental.json, 2024.

[105] Intel. The Intel Xeon 6 Processor Family. https://www.content.shi.com/cms-content/accelerator/media/pdfs/intel/intel-012026-intel-xeon6-productbrief.pdf, 2026.

[106] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *ACM SOSP*, 2009.

[107] Ravi Iyer. CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms. In *ACM ICS*, 2004.

[108] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. QoS Policies and Architecture for Cache/Memory in CMP Platforms. In *ACM SIGMETRICS PER*, 2007.

[109] Akanksha Jain, Hannah Lin, Carlos Villavieja, Baris Kasikci, Chris Kennelly, Milad Hashemi, and Parthasarathy Ranganathan. Limoncello: Prefetchers for Scale. In *ACM ASPLOS*, 2024.

[110] Raj Jain. Congestion Control and Traffic Management in ATM Networks: Recent Advances and a Survey. In *Computer Networks and ISDN systems*, 1996.

[111] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a Globally-Deployed Software Defined WAN. In *ACM SIGCOMM*, 2013.

[112] Min Kyu Jeong, Mattan Erez, Chander Sudanthi, and Nigel Paver. A QoS-Aware Memory Controller for Dynamically Balancing GPU and CPU Bandwidth Use in an MPSoC. In *IEEE/ACM DAC*, 2012.

[113] Djordje Jevdjic, Gabriel H Loh, Cansu Kaynak, and Babak Falsafi. Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache. In *IEEE/ACM ISCA*, 2014.

[114] Marty Kalin. CFS: Completely Fair Process Scheduling in Linux. `https://opensource.com/article/19/2/fair-scheduling-linux`, 2019.

[115] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. HeteroOS: OS Design for Heterogeneous Memory Management in Datacenter. In *IEEE/ACM ISCA*, 2017.

[116] Harshad Kasture and Daniel Sanchez. Ubik: Efficient Cache Sharing with Strict QoS for Latency-Critical Workloads. In *ACM ASPLOS*, 2014.

[117] Onur Kayiran, Nachiappan Chidambaram Nachiappan, Adwait Jog, Rachata Ausavarungnirun, Mahmut T Kandemir, Gabriel H Loh, Onur Mutlu, and Chita R Das. Managing GPU Concurrency in Heterogeneous Architectures. In *IEEE/ACM MICRO*, 2014.

[118] Anurag Khandelwal, Yupeng Tang, Rachit Agarwal, Aditya Akella, and Ion Stoica. Jiffy: Elastic Far-Memory for Stateful Serverless Analytics. In *ACM EuroSys*, 2022.

[119] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. Exploring the Design Space of Page Management for Multi-Tiered Memory Systems. In *USENIX ATC*, 2021.

[120] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *ACM PACT*, 2004.

[121] Yoongu Kim. Architectural Techniques to Enhance DRAM Scaling. `https://kilthub.cmu.edu/articles/thesis/Architectural_Techniques_to_Enhance_DRAM_Scaling/7461695/1`, 2015.

[122] Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers. In *IEEE HPCA*, 2010.

[123] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In *IEEE/ACM MICRO*, 2010.

[124] Leonard Kleinrock. Queueing Systems, Volume 1: Theory, 1975.

[125] Aneesh KV Kumar. [PATCH v7 00/12] mm/demotion: Memory Tiers and Demotion. `https://lwn.net/ml/linux-kernel/20220622082513.467538-1-aneesh.kumar@linux.ibm.com/`, 2022.

[126] HT Kung and Alan Chapman. The FCVC (Flow-Controlled Virtual Channels) Proposal for ATM Networks: A Summary. In *IEEE ICNP*, 1993.

[127] NT Kung and Robert Morris. Credit-Based Flow Control for ATM Networks. In *IEEE Network*, 1995.

[128] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *USENIX OSDI*, 2012.

[129] Chester Lam. A Look into Intel Xeon 6's Memory Subsystem. `https://chipsandcheese.com/p/a-look-into-intel-xeon-6s-memory`, 2025.

[130] Chang Joo Lee, Veynu Narasiman, Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt. DRAM-Aware Last-Level Cache Writeback: Reducing Write-Caused Interference in Memory Systems. `https://utw10235.utweb.utexas.edu/people/cjlee/TR-HPS-2010-002.pdf`, 2010.

[131] Donghyuk Lee, Lavanya Subramanian, Rachata Ausavarungnirun, Jongmoo Choi, and Onur Mutlu. Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM. In *ACM PACT*, 2015.

[132] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. MEMTIS: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination. In *ACM SOSP*, 2023.

[133] Baptiste Lepers, Vivien Quema, and Alexandra Fedorova. Thread and Memory Placement on NUMA Systems: Asymmetry Matters. In *USENIX ATC*, 2015.

[134] Baptiste Lepers and Willy Zwaenepoel. Johnny Cache: The End of DRAM Cache Conflicts (in Tiered Main Memory Systems). In *USENIX OSDI*, 2023.

[135] Philip Levis, Kun Lin, and Amy Tai. A Case Against CXL Memory Pooling. In *ACM HotNets*, 2023.

[136] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *ACM ASPLOS*, 2023.

[137] Qiang Li, Qiao Xiang, Derui Liu, Yuxin Wang, Haonan Qiu, Xiaoliang Wang, Jie Zhang, Ridi Wen, Haohao Song, Gexiao Tian, Chenyang Huang, Lulu Chen, Shaozong Liu, Yaohui Wu, Zhiwu Wu, Zicheng Luo, Yuchao Shao, Chao Han, Zhongjie Wu, Jianbo Dong, Zheng Cao, Jinbo Wu, Jiwu Shu, and Jiesheng Wu. From RDMA to RDCA: Toward High-Speed Last Mile of Data Center Networks Using Remote Direct Cache Access. `https://arxiv.org/abs/2211.05975`, 2023.

[138] Qiang Li, Qiao Xiang, Yuxin Wang, Haohao Song, Ridi Wen, Wenhui Yao, Yuanyuan Dong, Shuqi Zhao, Shuo Huang, Zhaosheng Zhu, Huayong Wang, Shanyang Liu, Lulu Chen, Zhiwu Wu, Haonan Qiu, Derui Liu, Gexiao Tian, Chao Han, Shaozong Liu, Yaohui Wu, Zicheng Luo, Yuchao Shao, Junping Wu, Zheng Cao, Zhongjie Wu, Jiaji Zhu, Jinbo Wu, Jiwu Shu, and Jiesheng Wu. More Than Capacity: Performance-Oriented Evolution of Pangu in Alibaba. In *USENIX FAST*, 2023.

[139] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. HPCC: High Precision Congestion Control. In *ACM SIGCOMM*, 2019.

[140] Haikun Liu and Bingsheng He. Reciprocal Resource Fairness: Towards Cooperative Multiple-Resource Fair Sharing in IaaS Clouds. In *IEEE/ACM SC*, 2014.

[141] Jinshu Liu, Hamid Hadian, Yuyue Wang, Daniel S Berger, Marie Nguyen, Xun Jian, Sam H Noh, and Huaicheng Li. Systematic CXL Memory Characterization and Performance Analysis at Scale. In *ACM ASPLOS*, 2025.

[142] Kefei Liu, Zhuo Jiang, Jiao Zhang, Haoran Wei, Xiaolong Zhong, Lizhuang Tan, Tian Pan, and Tao Huang. Hostping: Diagnosing Intra-Host Network Bottlenecks in RDMA Servers. In *USENIX NSDI*, 2023.

[143] Gabriel H Loh and Mark D Hill. Efficiently Enabling Conventional Block Sizes for Very Large Die-Stacked DRAM Caches. In *IEEE/ACM MICRO*, 2011.

[144] Chengzi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. Imbalance in the Cloud: An Analysis on Alibaba Cluster Trace. In *IEEE BigData*, 2017.

[145] Tao Luo, Mingen Pan, Pierre Tholoniat, Asaf Cidon, Roxana Geambasu, and Mathias Lécuyer. Privacy Budget Scheduling. In *USENIX OSDI*, 2021.

[146] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and Efficient GPU Cluster Scheduling. In *USENIX NSDI*, 2020.

[147] Zoltan Majo and Thomas R Gross. Memory System Performance in a NUMA Multicore Multiprocessor. In *ACM SYSTOR*, 2011.

[148] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *ACM SIGMOD*, 2010.

[149] Ilias Marinos, Robert NM Watson, Mark Handley, and Randall R Stewart. Disk|Crypt|Net: Rethinking the Stack for High-Performance Video Streaming. In *ACM SIGCOMM*, 2017.

[150] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *ACM ASPLOS*, 2023.

[151] John McCalpin. The Evolution of Single-Core Bandwidth in Multicore Systems. https://sites.utexas.edu/jdm4372/2023/12/19/the-evolution-of-single-core-bandwidth-in-multicore-systems-update/, 2023.

[152] John D McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. https://www.cs.virginia.edu/stream/, 1995.

[153] John D. McCalpin. Mapping Addresses to L3/CHA Slices in Intel Processors. https://repositories.lib.utexas.edu/items/78ed399f-0e5e-41fe-96e1-c12a5acf74d7, 2021.

[154] John D McCalpin. Bandwidth Limits in the Intel Xeon Max (Sapphire Rapids with HBM) Processors. In *ISC High Performance*, 2023.

[155] Paul E McKenney. Stochastic Fairness Queueing. In *IEEE INFOCOM*, 1990.

[156] Microsoft. Azure Delivers the First Cloud VM with Intel Xeon 6 and CXL Memory - Now in Private Preview. https://techcommunity.microsoft.com/blog/SAPApplications/azure-delivers-the-first-cloud-vm-with-intel-xeon-6-and-cxl-memory---now-in-priv/4470067, 2025.

[157] Timothy Prickett Morgan. CXL and Gen-Z Iron Out a Coherent Interconnect Strategy. https://www.nextplatform.com/2020/04/03/cxl-and-gen-z-iron-out-a-coherent-interconnect-strategy/, 2020.

[158] Thomas Moscibroda and Onur Mutlu. Distributed Order Scheduling and Its Application to Multi-Core DRAM Controllers. In *ACM PODC*, 2008.

[159] Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, and Thomas Moscibroda. Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning. In *IEEE/ACM MICRO*, 2011.

[160] Onur Mutlu. Memory Scaling: A Systems Architecture Perspective. In *IEEE IMW*, 2013.

[161] Onur Mutlu and Thomas Moscibroda. Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems. In *USENIX Security*, 2007.

[162] Onur Mutlu and Thomas Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *IEEE/ACM MICRO*, 2007.

[163] Onur Mutlu and Thomas Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems. In *IEEE/ACM ISCA*, 2008.

[164] Deepak Narayanan, Fiodar Kazhamiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen Boyd, and Matei Zaharia. Solving Large-Scale Granular Resource Allocation Problems Efficiently with POP. In *ACM SOSP*, 2021.

[165] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *USENIX OSDI*, 2020.

[166] Kyle J Nesbit, Nidhi Aggarwal, James Laudon, and James E Smith. Fair Queuing Memory Systems. In *IEEE/ACM MICRO*, 2006.

[167] Kyle J Nesbit, James Laudon, and James E Smith. Virtual Private Caches. In *IEEE/ACM ISCA*, 2007.

[168] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W Moore. Understanding PCIe Performance for End Host Networking. In *ACM SIGCOMM*, 2018.

[169] Khang Nguyen. Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family. `https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html?wapkw=intel%20cat`, 2016.

[170] Noam Nisan and Amir Ronen. Algorithmic Mechanism Design. In *Games and Economic Behavior*, 2001.

[171] Misja Nuyens and Adam Wierman. The Foreground–Background Queue: A Survey. In *Performance Evaluation*, 2008.

[172] NVIDIA. NVLink-C2C: Chip Interconnect Technology. `https://www.nvidia.com/en-us/data-center/nvlink-c2c/`, 2024.

[173] Yueyang Pan, Yash Lala, Musa Unal, Yujie Ren, Seung-seob Lee, Abhishek Bhattacharjee, Anurag Khandelwal, and Sanidhya Kashyap. Scalable Far Memory: Balancing Faults and Evictions. In *ACM SOSP*, 2025.

[174] Abhay K Parekh and Robert G Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-Node Case. In *IEEE/ACM TON*, 1993.

[175] Dylan Patel, Jeff Koch, Tanj Bennett, and Wega Chu. The Memory Wall: Past, Present, and Future of DRAM. `https://www.semianalysis.com/p/the-memory-wall`, 2024.

[176] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters. In *ACM EuroSys*, 2018.

[177] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security*, 2016.

[178] Michael Piatek, Tomas Isdal, Thomas Anderson, Arvind Krishnamurthy, and Arun Venkataramani. Do Incentives Build Robustness in BitTorrent. In *USENIX NSDI*, 2007.

[179] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. FairCloud: Sharing the Network in Cloud Computing. In *ACM SIGCOMM*, 2012.

[180] Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C Mogul, Yoshio Turner, and Jose Renato Santos. ElasticSwitch: Practical Work-Conserving Bandwidth Guarantees for Cloud Computing. In *ACM SIGCOMM*, 2013.

[181] Qifan Pu and Haoyuan Li. FairRide: Near-Optimal, Fair Cache Sharing. In *USENIX NSDI*, 2016.

[182] Yifan Qiao, Chenxi Wang, Zhenyuan Ruan, Adam Belay, Qingda Lu, Yiying Zhang, Miryung Kim, and Guoqing Harry Xu. Hermit: Low-Latency, High-Throughput, and Transparent Remote Memory via Feedback-Directed Asynchrony. In *USENIX NSDI*, 2023.

[183] Moin Qureshi and Gabriel H Loh. Fundamental Latency Trade-Offs in Architecturing DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design. In *IEEE/ACM MICRO*, 2012.

[184] Abishek Ramdas, David Cock, Timothy Roscoe, and Gustavo Alonso. The Enzian Coherent Interconnect (ECI): Opening a Coherence Protocol to Research and Applications. In *LATTE*, 2021.

[185] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *ACM SOSP*, 2021.

[186] Redis. Redis. `http://www.redis.io`.

[187] Redis. Redis Benchmark. `https://redis.io/docs/management/optimization/benchmarks/`, 2024.

[188] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *ACM SoCC*, 2012.

[189] Tim Roughgarden. Algorithmic Game Theory. In *ACM CACM*, 2010.

[190] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. AIFM: High-Performance, Application-Integrated Far Memory. In *USENIX OSDI*, 2020.

[191] Hugo Sadok, Miguel Elias M. Campista, and Luís Henrique M. K. Costa. Stateful DRF: Considering the Past in a Multi-Resource Allocation. In *IEEE TC*, 2021.

[192] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. `https://arxiv.org/abs/2003.03423`, 2020.

[193] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *USENIX OSDI*, 2018.

[194] Debendra Das Sharma. Compute Express Link (CXL): Enabling Heterogeneous Data-Centric Computing with Heterogeneous Memory Hierarchy. In *IEEE Micro*, 2023.

[195] Alan Shieh, Srikanth Kandula, Albert G Greenberg, and Changhoon Kim. Seawall: Performance Isolation for Cloud Datacenter Networks. In *USENIX HotCloud*, 2010.

[196] Shigeru Shiratake. Scaling and Performance Challenges of Future DRAM. In *IEEE IMW*, 2020.

[197] Madhavapeddi Shreedhar and George Varghese. Efficient Fair Queueing Using Deficit Round Robin. In *ACM SIGCOMM*, 1995.

[198] David Shue, Michael J. Freedman, and Anees Shaikh. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *USENIX OSDI*, 2012.

[199] Jaewoong Sim, Gabriel H Loh, Hyesoon Kim, Mike OConnor, and Mithuna Thottethodi. A Mostly-Clean DRAM Cache for Effective Hit Speculation and Self-Balancing Dispatch. In *IEEE/ACM MICRO*, 2012.

[200] Bryan Spry, Nagi Aboulenein, and Steve Kulick. United States Patent Application Publication: Mechanism for Write Optimization to a Memory Device. https://patentimages.storage.googleapis.com/53/bf/04/667faa6c4e5278/US20080162799A1.pdf, 2008.

[201] Ion Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy K Baruah, Johannes E Gehrke, and C Greg Plaxton. A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems. In *IEEE RTSS*, 1996.

[202] Michael Stonebraker and Ariel Weisberg. The VoltDB Main Memory DBMS. In *IEEE DEBull*, 2013.

[203] J. Stuecheli, W. J. Starke, J. D. Irish, L. B. Arimilli, D. Dreps, B. Blaner, C. Wollbrink, and B. Allison. IBM POWER9 Opens Up a New Era of Acceleration Enablement: OpenCAPI. In *IBM JRD*, 2018.

[204] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Ana Klimovic, Adrian Schuepbach, and Bernard Metzler. Unification of Temporary Storage in the NodeKernel Architecture. In *USENIX ATC*, 2019.

[205] Lavanya Subramanian, Donghyuk Lee, Vivek Seshadri, Harsha Rastogi, and Onur Mutlu. The Blacklisting Memory Scheduler: Achieving High Performance and Fairness at Low Cost. In *IEEE ICCD*, 2014.

[206] Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu. The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory. In *IEEE/ACM MICRO*, 2015.

[207] Lavanya Subramanian, Vivek Seshadri, Yoongu Kim, Ben Jaiyen, and Onur Mutlu. MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems. In *IEEE HPCA*, 2013.

[208] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. In *IEEE/ACM MICRO*, 2023.

[209] Shanjiang Tang, Bu-Sung Lee, Bingsheng He, and Haikun Liu. Long-Term Resource Fairness: Towards Economic Fairness on Pay-as-you-use Computing Systems. In *ACM ICS*, 2014.

[210] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. ResQ: Enabling SLOs in Network Function Virtualization. In *USENIX NSDI*, 2018.

[211] James Tuck, Luis Ceze, and Josep Torrellas. Scalable Cache Miss Handling for High Memory-Level Parallelism. In *IEEE/ACM MICRO*, 2006.

[212] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In *ACM ASPLOS*, 1996.

[213] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-Scale Cluster Management at Google with Borg. In *ACM EuroSys*, 2015.

[214] Midhul Vuppalapati and Rachit Agarwal. Tiered Memory Management: Access Latency Is the Key! In *ACM SOSP*, 2024.

[215] Midhul Vuppalapati, Saksham Agarwal, Henry Schuh, Baris Kasikci, Arvind Krishnamurthy, and Rachit Agarwal. Understanding the Host Network. In *ACM SIGCOMM*, 2024.

[216] Midhul Vuppalapati, Kushal Babel, Anurag Khandelwal, and Rachit Agarwal. SHORTSTACK: Distributed, Fault-Tolerant, Oblivious Data Access. In *USENIX OSDI*, 2022.

[217] Midhul Vuppalapati, Giannis Fikioris, Rachit Agarwal, Asaf Cidon, Anurag Khandelwal, and Éva Tardos. Karma: Resource Allocation for Dynamic Demands. In *USENIX OSDI*, 2023.

[218] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Snowflake Dataset Github Repository. `https://github.com/resource-disaggregation/snowset`.

[219] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building an Elastic Query Engine on Disaggregated Storage. In *USENIX NSDI*, 2020.

[220] Carl A Waldspurger. Memory Resource Management in VMware ESX Server. In *ACM OSR*, 2002.

[221] Carl A Waldspurger and William E Weihl. Stride Scheduling: Deterministic Proportional Share Resource Management. https://hdl.handle.net/1721.1/149244, 1995.

[222] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A Memory-Disaggregated Managed Runtime. In *USENIX OSDI*, 2020.

[223] Chenxi Wang, Haoran Ma, Shi Liu, Yifan Qiao, Jonathan Eyolfson, Christian Navasca, Shan Lu, and Guoqing Harry Xu. MemLiner: Lining Up Tracing and Application for a Far-Memory-Friendly Runtime. In *USENIX OSDI*, 2022.

[224] David Tawei Wang. Modern DRAM Memory Systems: Performance Analysis and Scheduling Algorithm, 2005.

[225] Hao Wang, Chang-Jae Park, Gyung-su Byun, Jung Ho Ahn, and Nam Sung Kim. Alloy: Parallel-Serial Memory Channel Architecture for Single-Chip Heterogeneous Processor Systems. In *IEEE HPCA*, 2015.

[226] Minhu Wang, Mingwei Xu, and Jianping Wu. Understanding I/O Direct Cache Access Performance for End Host Networking. In *ACM SIGMETRICS*, 2022.

[227] Xiaodong Wang and José F Martínez. XChange: A Market-Based Approach to Scalable Dynamic Multi-Resource Allocation in Multicore Architectures. In *IEEE HPCA*, 2015.

[228] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, et al. TMO: Transparent Memory Offloading in Datacenters. In *ACM ASPLOS*, 2022.

[229] Rich Wolski, John Brevik, Ryan Chard, and Kyle Chard. Probabilistic Guarantees of Execution Duration for Amazon Spot Instances. In *IEEE/ACM SC*, 2017.

[230] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. Nomad: Non-Exclusive Memory Tiering via Transactional Page Migration. In *USENIX OSDI*, 2024.

[231] Di Xie, Ning Ding, Y Charlie Hu, and Ramana Kompella. The Only Constant Is Change: Incorporating Time-Varying Network Reservations in Data Centers. In *ACM SIGCOMM*, 2012.

[232] Gala Yadgar, Michael Factor, and Assaf Schuster. Cooperative Caching with Return on Investment. In *IEEE MSST*, 2013.

[233] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble Page Management for Tiered Memory Systems. In *ACM ASPLOS*, 2019.

[234] Juncheng Yang, Yao Yue, and K. V. Rashmi. A Large Scale Analysis of Hundreds of In-Memory Cache Clusters at Twitter. In *USENIX OSDI*, 2020.

[235] Wonsup Yoon, Jisu Ok, Jinyoung Oh, Sue Moon, and Youngjin Kwon. Dilos: Do Not Trade Compatibility for Performance in Memory Disaggregation. In *ACM EuroSys*, 2023.

[236] George Yuan and Tor Aamodt. A Hybrid Analytical DRAM Performance Model. In *IEEE/ACM MoBS*, 2009.

[237] Yifan Yuan, Mohammad Alian, Yipeng Wang, Ren Wang, Ilia Kurakin, Charlie Tai, and Nam Sung Kim. Don't Forget the I/O When Allocating Your LLC. In *IEEE/ACM ISCA*, 2021.

[238] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *ACM EuroSys*, 2010.

[239] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *USENIX HotCloud*, 2010.

[240] David Zeng and Alexandros Psomas. Fairness-Efficiency Tradeoffs in Dynamic Fair Division. In *ACM EC*, 2020.

[241] Hui Zhang and Jon CR Bennett. WF2Q: Worst-Case Fair Weighted Fair Queueing. In *IEEE INFOCOM*, 1996.

[242] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. A Permutation-Based Page Interleaving Scheme to Reduce Row-Buffer Conflicts and Exploit Data Locality. In *IEEE/ACM MICRO*, 2000.

[243] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. Understanding Data Storage and Ingestion for Large-Scale Deep Recommendation Model Training: Industrial Product. In *IEEE/ACM ISCA*, 2022.

[244] Liang Zheng, Carlee Joe-Wong, Christopher G Brinton, Chee Wei Tan, Sangtae Ha, and Mung Chiang. On the Viability of a Cloud Virtual Service Provider. In *ACM SIGMETRICS*, 2016.

[245] Liang Zheng, Carlee Joe-Wong, Chee Wei Tan, Mung Chiang, and Xinyu Wang. How to Bid the Cloud. In *ACM SIGCOMM*, 2015.

[246] Yuhong Zhong, Daniel S Berger, Carl Waldspurger, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D Hill, Mosharaf Chowdhury, and Asaf Cidon. Managing Memory Tiers with CXL in Virtualized Environments. In *USENIX OSDI*, 2024.

[247] Yang Zhou, Hassan MG Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E Culler, Henry M Levy, et al. Carbink: Fault-Tolerant Far Memory. In *USENIX OSDI*, 2022.

[248] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion Control for Large-Scale RDMA Deployments. In *ACM SIGCOMM*, 2015.