UNIVERSITY OF CALIFORNIA

Los Angeles

Reinventing Datacenter System Stacks for Resource Harvesting

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Yifan Qiao

2024

ABSTRACT OF THE DISSERTATION

Reinventing Datacenter System Stacks for Resource Harvesting

by

Yifan Qiao
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2024
Professor Harry Guoqing Xu, Co-Chair
Professor Miryung Kim, Co-Chair

The rise of cloud computing and recent AI breakthroughs have radically expanded the demand for datacenter hardware resources, including CPU, memory, and accelerators such as GPUs. Despite the critical need to improve resource utilization and reduce operational cost, current datacenter system stacks—comprising OSes and runtime systems—struggle to fully utilize hardware resources due to high load variability and stringent performance requirements of datacenter workloads, leading to substantial waste of compute and memory resources.

This dissertation demonstrates that it is feasible to safely and efficiently harvest stranded datacenter resources, even when they are intermittently available and dispersed across servers. Specifically, we identify two previously overlooked resource harvesting opportunities in today's data center system stacks. First, although datacenter applications often have varying and potentially large resource demands, they typically include elastic components that can be safely discarded under resource pressure, making them ideal for utilizing idle resources with temporal availability. Existing operating systems and runtime systems, though, lack proper interfaces for applications to convey such semantics and take advantage of idle resources.

Second, while the availability of resources per server is unpredictable, combining stranded resources across servers can offer better overall availability. However, this opportunity is unavailable to many datacenter workloads that were designed for running on a single machine.

Driven by these insights, this dissertation rethinks the datacenter system stack and introduces holistic designs for OS abstractions, the OS kernel, and application runtime systems for resource harvesting. The contributions of this dissertation are fourfold.

First, we investigate how to harvest resources, especially memory which is inelastic and hard to re-assign between applications, within a single server. We introduce Midas, an OS memory abstraction that allows applications to use idle memory for storing their soft state. Midas efficiently manages soft memory with a kernel-runtime co-design, achieving near-optimal performance for four real-world datacenter applications and responding to extreme memory pressure quickly enough to avoid running out of memory.

Second, we explore how to harvest resources across servers. We present Hermit, a redesigned OS kernel paging/swap system that enables applications to harvest idle memory on remote servers with full transparency and efficiency. Hermit allows any application to harness remote memory without changing a single line of code, making it practical for legacy real-world datacenter applications. It also achieves three orders of magnitude lower tail latency and up to 1.87 times higher throughput for latency-critical and batch-processing applications, respectively.

Third, built atop Hermit, Canvas is a resource isolation mechanism for the kernel swap system that allows multiple applications to share remote memory without performance interference. By segregating resource usages and access patterns of co-running applications, Canvas further adaptively optimizes kernel swap for each application. Our evaluation and performance study with a wide range of datacenter applications demonstrate that Canvas reduces performance variation by a factor of 7 and improves their throughput by an average of 3.5 times when multiple applications share remote memory.

Finally, we demonstrate that our insights can be generalized to accelerators and emerging

AI workloads. We develop Concerto, a preemptive GPU runtime for large language model serving that harnesses idle GPU resources for offline inference tasks. By opportunistically batching offline inference tasks when online serving cannot fully saturate GPUs, Concerto significantly increases GPU utilization by an average of 2.35 times. By reactively preempting offline tasks upon online load bursts, Concerto reduces online serving latency by two orders of magnitude.

Together, these systems form a new datacenter system stack that synergistically enhances performance, resource utilization, and cost efficiency, offering a transformative approach to modern datacenter management.

The dissertation of Yifan Qiao is approved.

Adam Belay

Yuval Tamir

Miryung Kim, Committee Co-Chair

Harry Guoqing Xu, Committee Co-Chair

University of California, Los Angeles

2024

Dedicated to my family for their love and support

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Five years later, as I look back, I still remember clearly that distant afternoon when I started my PhD at UCLA. That moment marked the beginning of an incredible journey, filled with challenges, growth, and the unwavering support of many remarkable individuals who guided me along the way.

First and foremost, I would like to express my deepest gratitude to my advisors, Harry Xu and Miryung Kim. Their insightful guidance, encouragement, and patience have been the cornerstone of my academic journey. I remember the first time I talked to Harry when I was still an undergraduate, debating what to work on next. During that conversation, Harry advised me, *"Do not be trapped by what you have known. Always first find the most important problem to work on, then keep learning to solve it."* From that moment, I felt my research journey truly began. It turns out the advice was indeed true, and Harry has proven to be an incredible advisor. Harry not only directed me toward the important problems but also showed me how to transform seemingly wild ideas into concrete projects and accomplish them with solid algorithms, designs, and implementations. His patience, leadership, and integrity make him not only an exceptional advisor but also a mentor and a trusted friend to his students.

I am also sincerely thankful to Miryung, who agreed to co-advise me at the start of my PhD and has consistently supported me since. During my first visit to UCLA, Miryung shared her advising philosophy—*"Focus on the students, since graduating great students means you'll produce great research, while focusing on the research may or may not produce great students."* This philosophy has been evident throughout my time under her guidance. Miryung is a great advisor who spends tremendous time with students and always gives insightful advice on research, career planning, and personal development. Harry and Miryung were not only supportive during my entire PhD study but also generously gave me the freedom to pursue my own research interests and collaborate with various groups. Their distinct and unique

qualities have collectively formed the role model I aspire to emulate. All of these led to five papers across a wide range of topics, this PhD dissertation on resource harvesting systems, and five wonderful years of research and life with the best advisors possible.

I would also like to extend my gratitude to my committee members, Adam Belay and Yuval Tamir, for their valuable feedback, which has greatly helped me improve and complete this dissertation. A special thanks to Adam, who not only collaborated with me on various projects that contributed significantly to this dissertation but also hosted me for two summers at MIT. Adam is a true expert in operating systems and I am always impressed by his unique combination of comprehensive knowledge, deep insights, and creative innovations. Although we worked together on only two papers, they turned out to be some of my best work. Thank you, Adam, for introducing me to classic designs in operating systems and networks, unveiling the complexities of advanced CPU features, and demonstrating how one can be a passionate system researcher and a professional professor.

I have been very fortunate to be surrounded by such amazing people in the UCLA SOLAR lab: Haoran Ma, Chenxi Wang, John Thorpe, Shi Liu, Shan Yu, Shu Anzai, Christian Navasca, Jonathan Eyolfson, Arthi Padmanabhan, Pengzhan Zhao, Usama Hameed, Zhenting Zhu, Jiyuan Wang, Qian Zhang, Jason Teoh, Ben Limpanukorn, Yaoxuan Wu, Eric Zitong Zhou, Fabrice Harel-Canada, Burak Yetistiren, Hong Jin Kang, Yuanqi Li, Shen Teng, Gaohong Liu, Ricky Fok, among many others. Thank you all for the countless enriching research discussions, enjoyable gatherings, lasting friendships, and empathetic support we shared on this academic path. The bond we have formed is beyond words, and I am confident it will stand the test of time.

I would also like to express my gratitude to my undergraduate advisors. I am grateful to Jidong Zhai for introducing me to the world of system research. From Jidong, I have gained real experience in building and debugging high-performance systems and hardware ranging from supercomputers to accelerators. I also want to thank Youyou Lu, who advised me during my final undergraduate year and guided my thesis, which was nominated as an

outstanding undergraduate thesis. From Youyou, I learned his deep insights into storage and memory systems and how to present in a clear and elegant fashion. The skills and insights I gained from Jidong and Youyou laid the foundation for my PhD journey and are reflected throughout this dissertation.

I am thankful to all my friends for both research and non-research discussions: Yang Wang, Shuyang Liu, Pradeep Dogga, Zeina Migeed, Sihao Liu, and Weitong Zhang at UCLA; Zhenyuang Ruan, Gohar Chaudhry, Inho Cho, Josh Fried, Yun-Sheng Chang, Sanjit Bhat, Ariel Szekely, Alexandra Henzinger, Derek Leung, and Upamanyu Sharma at MIT; Shen Wang, Rui Yang, Yonger Xue, Xiaojun Hu, and Sijie Chen at the Ohio State University; Yifan Qiao (yes, we have the same alphabetical name and, even more coincidentally, come from the same hometown) and Xuandong Zhao at UCSB; Qibin Chen at CMU, and many others—your support means a lot, and you know who you are. Your friendship has been a constant source of strength, providing much-needed balance and perspective through the ups and downs of PhD life.

Lastly, my heartfelt thanks go to my family for their unwavering support as I pursued my PhD. Special thanks to my father, Xiangyang, my mother, Hongyun, and my brother, Qifan, for their unconditional love and support throughout the past five years. I am eternally grateful for their presence through all my highs and lows. I hope to have made you all proud.

"Let this be my last word, that I trust in thy love."

Rabindranath Tagore

VITA

Graduate Student Researcher/Teaching Assistant University of California, Los Angeles	2019 - 2024
M.S. in Computer Science University of California, Los Angeles	2019 - 2022
B.S. in Computer Science and Technology Tsinghua University	2015 - 2019

PUBLICATIONS

Yifan Qiao, Zhenyuan Ruan, Haoran Ma, Adam Belay, Miryung Kim, Harry Xu, "Harvesting Idle Memory for Application-managed Soft State with Midas", NSDI, 2024

Haoran Ma, **Yifan Qiao**, Shi Liu, Shan Yu, Yuanjiang Ni, Qingda Lu, Jiesheng Wu, Yiying Zhang, Miryung Kim, Harry Xu, "DRust: Language-Guided Distributed Shared Memory with Fine Granularity, Full Transparency, and Ultra Efficiency", OSDI, 2024

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", NSDI, 2023

Chenxi Wang, **Yifan Qiao** (co-first author), Haoran Ma, Shi Liu, Yiying Zhang, Wenguang Chen, Ravi Netravali, Miryung Kim, Guoqing Harry Xu, "Canvas: Isolated and Adaptive Swapping for Multi-Applications on Remote Memory", NSDI, 2023

John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, **Yifan Qiao**, Zhihao Jia, Minjia Zhang, Ravi Netravali, Guoqing Harry Xu, “Bamboo: Making Preemptible Instances Resilient for Affordable Training of Large DNNs”, NSDI, 2023

Chenxi Wang, Haoran Ma, Shi Liu, **Yifan Qiao**, Jonathan Eyolfson, Christian Navasca, Shan Lu, Guoqing Harry Xu, “MemLiner: Lining up Tracing and Application for a Far-Memory-Friendly Runtime”, OSDI, 2022, **Best Paper Award**

Haoran Ma, Shi Liu, Chenxi Wang, **Yifan Qiao**, Michael D Bond, Stephen M Blackburn, Miryung Kim, Guoqing Harry Xu, “Mako: a low-pause, high-throughput evacuating collector for memory-disaggregated datacenters”, PLDI, 2022

John Thorpe, **Yifan Qiao** (co-first author), Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu, “Dorylus: Affordable, Scalable, and Accurate GNN Training over Billion-Edge Graphs”, OSDI, 2021

CHAPTER 1

Introduction

Over the past few decades, datacenters have greatly changed the landscape of computing. They become the key infrastructure behind cloud computing [20], artificial intelligence [86, 189, 243], and other services such as online shopping and social networks that affect our daily lives. As a result, cloud providers have seen unprecedentedly rapid growth in their demand for servers to build datacenters, incurring huge costs of server maintenance and energy. It is extremely important to improve resource utilization for datacenters, as even small improvements could be greatly amplified by the great number of servers in datacenters. It is estimated that even 1% utilization improvement could save around seven billion dollars and 140M kilowatt-hours annually [42, 88, 104, 253]. Despite the huge potential cost savings and carbon footprint reductions, improving resource utilization in datacenters remains a notoriously difficult challenge. As reported by major cloud providers such as Google, Azure, Alibaba, and Snowflake, the average CPU and memory utilization in their datacenters is only around 40–60% [78, 150, 246, 256]. The situation can be even worse for GPUs, whose utilization can be as low as 20–50% [150, 264].

One of the major causes of resource inefficiency is rooted in the datacenter workloads, which expose strict performance requirements and high load variability. For example, the load of Google's mail service can increase by more than two times during peak hours than in non-peak hours [19]. It is also reported that ChatGPT [189] can experience two orders of magnitude higher request rates during load bursts [263]. To avoid running out of resources during peak hours, datacenter operators often have to overprovision resources for the application's peak usage, causing severe resource waste.

One promising solution to improve datacenter utilization is to *harvest* resources by consolidating workloads and colocating another best-effort application using idle resources left by the performance-critical application [16, 76, 78, 97, 194, 212, 217, 218, 225]. Unlike performance-critical applications that have strict quality-of-service requirements and demand for high availability, best-effort applications usually come with loose or no quality-of-service requirement and can tolerate interruptions. Consequently, datacenter operators can run best-effort applications at a lower priority when idle resources are available, and kill or shrink them to free up resources when the server is overloaded. Typical examples of best-effort applications include batch-processing applications such as Spark [274] and machine-learning training frameworks such as PyTorch [197] and TensorFlow [9]. When the performance-critical application encounters a load burst, a cluster manager can reactively reassign resources to them by punishing and reclaiming resources from the best-effort applications. The resource reclamation can be cooperative such as letting the best-effort applications themselves checkpoint their states and exit, or done in an enforced manner such as leveraging the operating system kernel to swap out best-effort applications to second-tier storage. Later, when the load burst subsides, the victim application can resume by loading its checkpoint or swapping in its data, ensuring the server operates at maximum utilization.

In this dissertation, we focus on two critical yet challenging resources to harvest in datacenters: memory and GPUs, while leaving the discussion of other resources, such as CPU, network, and storage, in future work (§7.1). We argue that memory and GPUs not only power the most demanding modern workloads, such as web services, big data analytics, and AI models, but also represent significant cost and energy challenges in datacenter operations.

First, many traditional datacenter applications, such as web services, big data analytics, and machine learning systems, are in-memory workloads. To support them, datacenters must continuously expand their memory capacity to satisfy increasing demands. Meanwhile, with the slowing of Moore's law and Dennard Scaling, hardware vendors face challenges in improving DRAM storage density and reducing manufacturing costs per unit [132]. As a result, memory has become a significant driver of hardware costs and energy consumption in datacenters. The high cost of memory, coupled with its generally low utilization, suggests substantial potential for harvesting memory. Yet, memory is difficult to harvest. Unlike CPU cores, which can be overcommitted with multiple threads and reassigned between processes, memory is inherently *inelastic* once allocated and holding application data, making it challenging to reassign without disrupting application performance.

At the same time, recent advances in AI, particularly large language models, have spurred significant demand for accelerators like GPUs. While GPUs continue to improve performance to meet AI's high computational demands, they come with steep hardware and energy costs. High-end GPUs, such as the Nvidia H100 or AMD MI300X, can cost around $20,000 each, and modern datacenters often need thousands of such GPUs. Despite the urge need to make full use of GPUs, today's datacenters still suffer from low GPU utilization. This is due to the inherent challenges in harvesting GPU resources. GPU workloads typically have high compute demands and large memory footprints, making it difficult to co-locate multiple applications on the same device. Additionally, the lack of comprehensive virtualization and resource-sharing support for GPUs limits opportunities for overcommitment and resource reclamation, further reducing utilization efficiency.

1.1 Challenges

The fundamental challenge hindering the efficient harvesting of resources in existing system stacks is *the speed at which resources can be reassigned.* While it is acceptable for performance-critical applications to temporarily lend reserved but unused resources to the other applications, these resources must be reclaimed quickly upon load bursts to avoid violating performance requirements or worse still out-of-resource killing. And given that load of datacenter applications can ramp up quickly within seconds [76, 194, 204, 263], the underlying system stacks, including the application runtime and the operating system, must reclaim resources from the best-effort application and grant them to the demanding performance-critical application at the same or even finer timescales (*e.g.*, sub-second timescales). For example, ChatGPT, a popular large-language-model-based chatbot running on GPUs, can experience 10× load fluctuations in just a few seconds [263]. To serve all the requests, the chatbot needs to allocate proportional GPU memory to store the per-request state commonly referred to as the KV cache [193]. The underlying GPU runtime hence must repartition the GPU and free up enough GPU memory fast enough (*i.e.*, within seconds), or else the incoming requests will be blocked and quickly queued up, leading to catastrophic tail latency degradation. The situation can be even worse for traditional workloads and resources such as memory. For example, microservices running on Linux can allocate 1 GiB memory within just 100 ms [208, 218], and they will get killed if the OS kernel cannot reclaim enough memory at the same speed from best-effort applications before the server runs out of memory.

Unfortunately, such a strict speed requirement for reassigning resources is far beyond the capability of existing datacenter system stacks, which can only reclaim resources at the timescale of seconds or even minutes. Traditionally, the OS kernel leverages memory paging, a classic idea that dates back to the 1960s, to swap overcommitted memory pages to secondary storage to resolve memory pressure. However, due to the limited bandwidth of the storage and software overheads of the kernel swap system, kernel paging is orders of magnitude slower than the memory allocation speed. More recently, people have been investigating more flexible resource harvesting techniques at the virtual machine (VM) level. Major cloud providers now offer spot VMs [15, 87, 164], which co-locate with regular VMs but run at a low priority to utilize vacant resources. Spot VMs are evicted under resource pressure to free up capacity. However, preempting a spot VM is *inefficient*, as the best-effort applications running on them may lose partially completed work, requiring them to restart the entire process. On the other hand, alternative VM designs, such as resource-harvesting VMs [16, 78, 212, 283] and deflatable VMs [225], support gracefully adjusting VMs' resource usage, thereby avoiding disruption to the best-effort applications running on them. Although these designs achieve higher efficiency, they do so at the expense of slower resource reassignment, often taking minutes to reconfigure a VM. This delay can be *unsafe* for performance-critical applications, as they may run out of resources during the reassignment process.

In summary, existing resource harvesting techniques struggle to balance efficiency and safety. This limitation stems from today's datacenter system stacks, originally designed for traditional applications with relatively static and stable loads, where resource reallocation was a rare event. As modern datacenter applications become increasingly dynamic with highly fluctuating loads, it is crucial to rethink the design principles of datacenter system stacks, prioritizing resource reassignment speed in order to make resource harvesting practical.

This dissertation aims to answer the following question: can we harvest stranded resources in datancenters with both efficiency and safety? With efficiency, the application should be able to proportionally trade additional resources for better performance, rather than being disruptively killed and wasting work it has already done. With safety, the application should be able to rapidly release and return previously harvested resources, so that it can get resources timely to preserve quality of service even under drastic load fluctuations.

1.2 Insights

This dissertation introduces a reinvented datacenter system stack based on two independent yet complementary insights.

First, existing datacenter system stacks are mainly capable of managing resources for applications' static components, but they lack interfaces and mechanisms to support elastic components—those that can benefit from additional resources when available but can be safely discarded without disrupting the application. For instance, many applications use caches to reduce disk or network traffic and improve performance [22, 30]. Caches are well-suited for idle memory, as they can be discarded during memory pressure and later regenerated from backend storage. By introducing appropriate abstractions and interfaces and mapping elastic components to idle resources, we can efficiently utilize vacant resources when available and safely release them under resource pressure.

Second, current datacenter system stacks primarily focus on reallocating resources within a single server, overlooking the potential of utilizing idle resources across multiple servers. With advancements in modern datacenter networks [137, 162, 219], it is now feasible to run applications using resources distributed across several servers [12, 89, 224]. This opens up a unique opportunity for applications to harvest remote resources and gradually shift their resource usage to other servers under pressure, rather than abruptly disrupting execution. Moreover, while resource availability on a single server can be unpredictable, aggregating resources across a cluster could offer greater stability and capacity. Consequently, enabling remote resource harvesting could significantly enhance overall resource utilization. However, this opportunity is not available to many datacenter applications, because they are designed to run on a single machine and require manual adaptation to function across multiple servers. Additionally, even for applications for which distribution is possible, the high OS kernel software overhead in cross-server communication can diminish the benefits of additional resources. We propose that by redesigning the application runtime and OS kernel, we can enable single-machine applications to transparently and efficiently utilize idle resources distributed across servers.

1.3 Dissertation Statement

It is feasible to safely and efficiently harvest idle resources in datacenters, even if they are only intermittently available and spread across servers. Specifically, this can be achieved with new interfaces that allow applications to specify their elastic components, OS/runtime co-designs that enable applications and the underlying operating system to co-manage idle resources, and semantics-guided operating system designs that improve efficiency and practicality by allowing applications to use idle resources on remote servers.

1.4 Contributions

This dissertation makes the following contributions:

OS Abstractions for efficient and safe local memory harvesting. We first identify the semantics gap between the application soft state, which can benefit from additional memory and be safely discarded, and the operating system kernel, which is in charge of memory management but unaware of the most beneficial data to spend idle memory on. To effectively manage and dynamically scale soft state, we propose *soft memory*, an elastic virtual memory abstraction with *unmap-and-reconstruct* semantics that makes it possible for applications to use idle memory to store whatever soft state they choose while guaranteeing both safety and efficiency. We present Midas, a soft memory management system that contains (1) a runtime that is linked to each application to manage soft memory objects and (2) OS kernel support that coordinates soft memory allocation between applications to maximize their performance. Our experiments with four real-world applications show that Midas can efficiently and safely harvest idle memory to store applications' soft state, delivering near-optimal application performance and responding to extreme memory pressure without running out of memory.

Transparent and Efficient OS kernel swapping for remote memory harvesting. We then explore techniques that enable applications to scale out beyond a single machine and harvest resources on remote servers. We identify the OS paging/swap system as a viable approach to practical remote memory harvesting. However, its current design, which originally targets slow hard drives, is ill-suited for today's fast datacenter networks and significantly hinders efficiency. We present Hermit, a redesigned swap system that overcomes inefficiencies and offers applications a transparent and efficient way to harvest remote memory. The core technique of Hermit is *adaptive, feedback-directed asynchrony*, which takes non-urgent but time-consuming operations (*e.g.*, swap-out, `cgroup` charge, I/O deduplication, *etc.*) on the fault-handling path and executes them asynchronously. Additionally, Hermit collects runtime feedback from applications and uses it to direct how asynchrony should be performed—*i.e.*, whether asynchronous operations should be enabled, the level of asynchrony, and how asynchronous operations should be scheduled. An evaluation with a set of latency-critical applications shows that Hermit delivers low-latency remote memory. For example, it reduces the 99th percentile latency of Memcached by 99.7% from 36 ms to 91 µs. Running Hermit over batch applications improves their overall throughput by up to 1.87×. These results are achieved without changing a single line of user code, demonstrating the feasibility of memory elasticity with transparent and efficient remote memory.

Isolated and adaptive remote memory harvesting for multi-applications. A critical step towards the practical deployment of remote memory harvesting systems is ensuring that multiple applications can share remote memory without interference. To address this challenge, we propose Canvas, a redesigned swap system that fully isolates swap paths for remote-memory applications. Canvas allows each application to possess its dedicated swap partition, swap cache, prefetcher, and RDMA bandwidth. Swap isolation further lays a foundation for adaptive optimization techniques based on each application's own access patterns and needs. We develop three such techniques: (1) adaptive swap entry allocation, (2) semantics-aware prefetching, and (3) two-dimensional RDMA scheduling. A thorough evaluation with a set of widely deployed applications demonstrates that Canvas minimizes performance variation and dramatically reduces performance degradation.

Large language model runtime system for GPU harvesting. Recent breakthroughs in artificial intelligence and large language models have significantly increased the demand for GPUs and accelerated computing. To democratize these AI advancements for a broader range of users and applications, it is crucial for datacenter system stacks to manage GPU resources for AI workloads efficiently to achieve both high performance and optimal GPU utilization. To address this need, we introduce Concerto, a large language model (LLM) runtime that harvests stranded GPU resources for *offline* LLM inference tasks such as document summarization and model evaluation. Unlike online inference tasks that are latency-critical, offline tasks usually run in a batch-processing fashion with much looser latency requirements, making them a good fit for stranded resources that are only available shortly. Concerto contains (1) an execution engine that preempts offline tasks timely in response to online load bursts, (2) an incremental checkpointing mechanism that minimizes the amount of recomputation required by preemptions, and (3) a scheduler that adaptively batches offline tasks with online tasks to maximize GPU utilization. Our evaluation demonstrates that Concerto not only achieves consistently low online serving latency but also enables a linear trade-off between online serving throughput and offline serving throughput without interference. As a result, Concerto improves the average GPU utilization by 2.35× and reduces 99^{th} percentile serving latency by 98.8%.

1.5 Dissertation Organizations

We organize this dissertation as follows. Chapter 2 first sets the context for our contributions with the necessary background on datacenter workloads and existing system stacks on improving resource utilization. The rest of the dissertation consists of four parts.

- **Part 1** deals with the memory inefficiency within a server due to the semantics gap between the application soft state and the OS kernel memory management. Chapter 3 presents Midas, a new OS virtual memory abstraction, and how the abstraction enables the coordination between the kernel and the runtime to harvest idle memory for application-managed soft state.

- **Part 2** includes the OS kernel redesign contributions of this dissertation, which enable applications to harvest idle memory on remote servers transparently and efficiently. Chapter 4 introduces Hermit, a redesigned OS kernel paging/swap system that allows applications to transparently scale out by efficiently swapping data to idle memory on remote servers, improving both application performance and overall resource utilization. Based on Hermit, Chapter 5 then discusses Canvas, an extension to kernel swap system that provides isolation support and adaptive optimizations for multi-application settings. In consequence, Canvas enables co-running applications to share remote memory without performance interference. More importantly, Canvas exploits the swap semantics of each isolated application, so that it can automatically adapt the swap system to the specific needs of each application to maximize hardware utilization and overall application performance.

- **Part 3** generalizes our methodology to harvest GPUs for AI workloads. Chapter 6 presents Concerto, a GPU runtime system that harvests idle GPU resources for offline large language model serving. Concerto colocates and schedules online and offline requests *within* the runtime and entirely eliminates the overheads associated with cluster re-partitioning and GPU reassignment in lower-level system stacks. This design enables Concerto to achieve low online latency, high offline throughput, and high GPU utilization simultaneously.

- **Part 4** describes future research directions and concludes the dissertation (Chapter 7).

Bibliograhpic Notes

Portions of this dissertation draw on work that I previously conducted and published with the guidance of my advisors Guoqing Harry Xu and Miryung Kim, and the support of other outstanding collaborators. Chapter 3 extends a conference paper published in NSDI '24 with Zhenyuan Ruan, Haoran Ma, Adam Belay, Miryung Kim, and Guoqing Harry Xu [208]. Chapter 4 is based on a conference paper published in NSDI '23 with Chenxi Wang, Zhenyuan Ruan, Adam Belay, Qingda Lu, Yiying Zhang, Miryung Kim, Guoqing Harry Xu [207]. Chapter 5 is based on a conference paper published in NSDI '23 with Chenxi Wang , Haoran Ma, Shi Liu, Yiying Zhang, Wenguang Chen, Ravi Netravali, Miryung Kim, Guoqing Harry

Xu [260]. Note that Canvas was co-authored with Chenxi Wang.

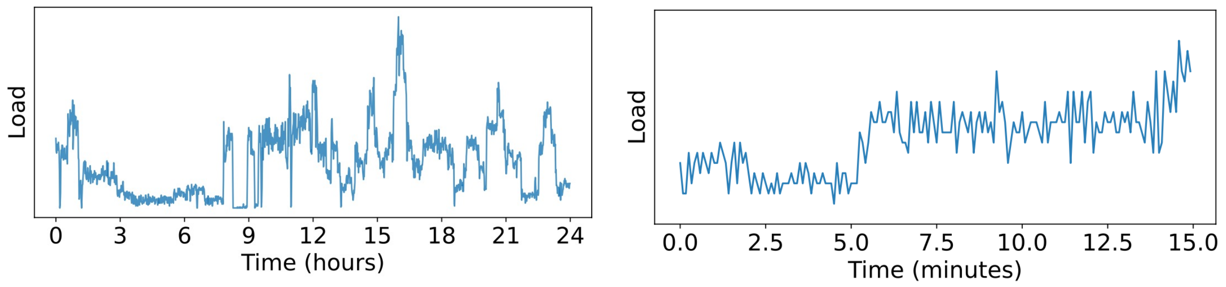
CHAPTER 2

Background

2.1 Characterizing Datacenter Workloads

With the prevailing cloud computing paradigm, modern datacenter applications are evolving into increasingly complex forms, involving larger scales, more stringent performance requirements, and heterogeneous resource demands. Typical examples of such applications include web search engines, social networks, online shopping platforms, and emerging big data analytics and machine learning frameworks. After years of widespread adoption, researchers and practitioners have identified three major characteristics of contemporary datacenter workloads, each posing unique challenges to datacenter system stack design.

Microsecond tail latency. Many datacenter applications, such as web search engines, social networks, and AI-powered chatbots, are essentially large-scale distributed systems. To ensure a smooth user experience, they must consistently perform well when serving millions or billions of requests. These systems are extremely sensitive to tail latencies, as even microsecond-scale outliers can significantly degrade service quality [64, 66]. The evolution of datacenter applications toward microservice architectures has further intensified this sensitivity. In microservice architectures, a single service could comprise dozens of or even hundreds of microservices that are distributed across multiple servers in a complex fan-in/fan-out pattern [54, 55, 80, 236]. Consequently, the overall response time of a user request is dictated by the slowest-performing microservice [64]. In practice, these microservices often have strict tail latency service-level objectives (SLOs), such as 99^{th} or 99.9^{th} percentile



(a) Load variation over 24 hours. (b) Load variation over 15 minutes.

Figure 2.1: User traffic to ChatGPT within a campus [263] exposes high load variability at both macro (2.1a) and micro (2.1b) time scales.

latency in microsecond timescale [26, 103, 157], to ensure consistently low end-to-end response latency.

Meeting these stringent tail latency requirements requires not only high-performance hardware but also highly efficient system stacks that operate at the same or even finer timescale [26]. For instance, if the resource harvesting system exploits idle resources left by a latency-critical application but fails to return them in time when needed, it can easily ramp up the application's tail latency by orders of magnitudes, leading to vast SLO violations that hurt user experience and the reliability of the entire service.

High load variability. Real-world datacenter applications exhibit significant load variability and diurnal patterns [19, 22, 194] which pose new challenges for efficient resource management. This variability arises from two primary factors. First, many datacenter applications are interactive and user-facing, meaning their load depends heavily on user activity. As a result, traffic peaks during busy hours and drops during off-peak periods. For example, Figure 2.1a illustrates how user traffic to ChatGPT [189] fluctuates over a 24-hour period within a university campus [263]. As seen, the load during the afternoon (between t = 15h and 18h) is nearly two orders of magnitudes higher than during the late night (between t = 1h and 7h).

However, load variability is not confined to macro timescales; it is also intrinsic at micro timescales, even within short time periods. Figure 2.1b zooms into a 15-minute window during peak hours to highlight significant micro load bursts and fluctuations. At time t = 5min and t = 9min, the load surges and then drops by more than two times. Similar patterns also recur frequently during time t = 10min−15min, where multiple load bursts occur.

These high levels of load variability and frequent micro bursts create significant challenges for datacenter resource management. System stacks must dynamically and quickly re-allocate resources in real-time to accommodate sudden load changes to prevent resource contention and maintain optimal application performance.

Heterogenous resource demands. Machine learning and artificial intelligence (AI) applications [9, 189, 197, 213, 247] are rapidly emerging as some of the most popular and resource-diverse workloads in datacenters. Unlike traditional workloads that primarily depend on CPU, memory, network, and storage resources, AI workloads heavily leverage specialized accelerators, such as GPUs, for their computational tasks. However, while CPUs and memory can be managed directly by the operating system and are readily accessible to applications, accelerators depend on vendor-specific drivers and runtimes. For instance, Nvidia GPUs rely on specialized drivers and the CUDA runtime, compelling developers to manually implement accelerated compute logic using CUDA libraries that are separate from their main application logic.

Additionally, on the hardware side, next-generation datacenter networks and storage devices [3, 36, 103, 133] are becoming increasingly configurable, offering internal channels and tuning knobs that allow applications to customize hardware for maximum performance. However, realizing the full potential of this type of configurable hardware requires a hardware-software co-design approach, where the application is rewritten with specialized runtimes and programming libraries such as DPDK [73] and SPDK [3].

In summary, the rise of AI workloads and the advent of configurable high-performance hardware present significant challenges for efficient resource management. To fully exploit these advancements, it is crucial for system stacks to manage heterogeneous applications and hardware resources with flexibility and efficiency, ensuring that the full potential of both software and hardware is unleashed.

2.2 Existing Datacenter System Stacks for Resource Harvesting

This section provides essential background on existing techniques for resource harvesting in datacenter system stacks. We categorize these techniques based on the specific layer of the system stack they operate within, as detailed in the following subsections.

2.2.1 Operating System Kernel Paging/Swap System

Historically, the OS kernel is designed to allow memory overcommitment with its virtual memory abstraction and the paging/swap mechanism. Paging was designed to extend the addressable memory space with a slow but large secondary storage (usually a mechanical disk). Under memory pressure, the kernel pages out cold pages to disk and marks them as absent from memory. Later, if a process accesses any of those pages, the memory management unit (MMU) raises a page fault exception which transparently traps the control flow into the kernel to page in the data and updates the corresponding page table entry (PTE). The process can then resume its execution and access the page normally. Ideally, with the support of paging, the OS kernel can support a best-effort application to run with limited physical memory left unused by the performance-critical application, and it can swap out the data of the best-effort application to return memory to the performance-critical application under load bursts, thereby achieving high overall utilization.

Linux implements paging in its swap subsystem, and Figure 2.2 illustrates the detailed stages when swapping in a page in Linux kernel. In Linux swap, the backend storage (*e.g.*, local disks for local swapping, and network-attached storage for remote swapping) is mapped

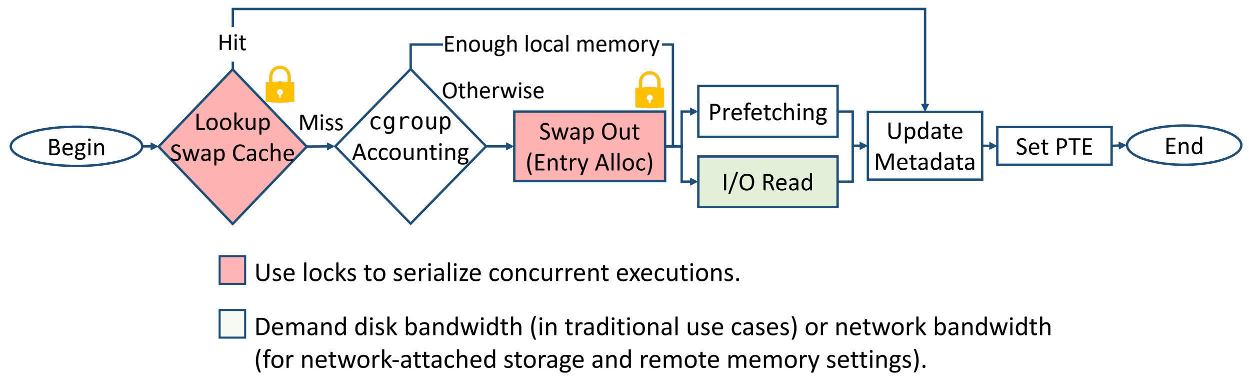

Begin
Hit
Lookup Swap Cache
Miss
Enough local memory
Otherwise
cgroup Accounting
Swap Out (Entry Alloc)
Prefetching
I/O Read
Update Metadata
Set PTE
End
🟥 Use locks to serialize concurrent executions.
🟩 Demand disk bandwidth (in traditional use cases) or network bandwidth (for network-attached storage and remote memory settings).

Figure 2.2: Stages that a swapped-in page undergoes in the Linux kernel's paging/swap system.

into the host server as a swap partition where application data will be swapped out. The swap partition is split into a set of 4KB *swap entries*, each mapping to an actual remote memory cell and has a unique entry ID. Upon a page fault, the kernel uses the swap entry ID contained in the corresponding page table entry (PTE) to locate the swap entry that stores the page.

The first step in handling the fault is to look up the swap cache, which serves as a centralized component that prevents race conditions. It uses a set of radix trees to track the information of swapped-in pages and ongoing swap-out requests at the granularity of a block (*e.g.*, 64MB) of swap entries. If a page can be found there, it gets mapped to a virtual page and removed from the swap cache. Otherwise, the kernel needs to perform a *demand swap-in*.

Before fetching the page, the kernel first does `cgroup` accounting to inspect if there is enough physical memory to swap in the page. If there is, the kernel issues an I/O read request. As the demand swap occurs, the kernel prefetches a number of pages that will likely be accessed in the future. This number depends on the swap history at the past few page faults. For example, if the pages fetched follow a sequential or a strided pattern, the kernel will use this pattern to fetch a few more pages. If no pattern is found, the kernel reduces the number of prefetched pages until it stops prefetching completely. Once these demand and prefetched pages arrive, they are placed into the swap cache. Their swap entries in remote memory are then freed.

If `cgroup` accounting deems that local memory is insufficient for the new page, the kernel must reclaim memory by evicting pages using the CLOCK algorithm [57]. Evicting a page *unmaps* it and pushes it into the swap cache. When memory runs low, the kernel releases existing pages from the swap cache to make room for newly fetched pages. Clean pages can be removed right away and dirty pages must be written back. To write back a page, the swap system must first allocate a swap entry using a free-list-based allocation algorithm. Then, an I/O write request is generated and the page is written into the newly allocated swap entry. Linux iteratively reclaims pages until the available memory rises above a low threshold.

Finally, Linux updates kernel metadata, including the page table entry (PTE) and the swap entry to mark the finish of the swap-in. During the entire swap-in process, extensive locking is needed for swap cache lookup, swap entry allocation, and metadata update—shared allocation metadata (*e.g.*, free list) must be protected when multiple applications/threads request swap entries simultaneously.

The architecture of the OS kernel swap has remained relatively stable since its inception, when disks were slow (10 ms, 200 MB/s). In the era of these slow disks, paging/swapping was typically used as a last resort to prevent out-of-memory (OOM) killing rather than as a viable way to harvest memory, as the swap rate is seriously limited by disk performance. However, today's advanced NVMe SSDs (20 µs, 8 GB/s) and fast datacenter networks (4 µs, 12 GB/s) offer significantly higher I/O throughput and bandwidth, which suggest the potential to revive paging as a practical solution for memory harvesting. Nevertheless, the design of today's swap systems, which originally targeted slow, disk-based storage, is rendered obsolete and incurs high software overheads. As will be discussed in Chapters 4 and 5, the legacy swap system software stack now becomes a critical bottleneck when modern datacenter applications attempt to leverage it for efficient memory harvesting, and we must redesign the swap system to align with current hardware capabilities to make memory harvesting practical.

2.2.2 Resource-Harvesting Runtime Systems

In addition to the OS kernel support, another line of work focuses on userspace runtime techniques to scale out applications for better resource utilization.

Distributed Shared Memory

Distributed shared memory (DSM) is one type of pioneer work on distributing an application to leverage available resources on remote servers. Its idea can be dated back to 1980s [38, 40, 41, 71, 138, 140, 165, 237], and there are still many active research efforts [34, 121, 153, 178, 223, 240, 261]. DSM runtimes provide applications with a unified, contiguous memory view, allowing them to run with distributed compute and memory resources. However, despite years of research efforts, DSM systems still struggle to achieve practical adoption due to their high communication and coherence overheads that stem from the need to maintain data consistency across distributed nodes. For example, a recent study [153] reported that even a modern DSM system with a 40Gbps network will still incur a 77% coherence overhead when accessing another server, which can slow down the application by 2.4×. Even worse, due to the potentially exponential communication complexity of the number of servers, DSM systems cannot scale to a large cluster, further hindering their adoption.

Resource-Disaggreged Runtimes

To mitigate extensive communication and synchronization costs, another line of work, exemplified by resource-disaggregated runtimes, trades off transparency for efficiency. The key idea behind resource disaggregation is to break the physical server boundary using fast network interconnections such as RDMA [162] and CXL [61, 137]. While this approach initially requires new datacenter architectures [69, 81], it can also be adapted to existing datacenters by abstracting resources across servers into logical pools (*e.g.*, logical memory pools [14] and logical CPU pools). This adaption enables applications to run with logically disaggregated resources, thereby improving resource utilization. To facilitate applications to operate with resources scattered across multiple servers, many resource-disaggregated programming frameworks and runtimes explicitly expose the physical separation of resources to the application, requiring developers to write code that is aware of the disaggregated architecture.

For example, memory-disaggregated programming frameworks like AIFM [216], Mira [95], and Atlas [44] introduce remote-able pointers and data structures to facilitate remote-memory-aware applications. Developers can use these programming primitives to rewrite applications, allowing them to offload specific data to remote memory. These frameworks' runtimes can then automatically manage data swapping to remote memory under pressure and retrieve it upon pointer dereference.

Compute-disaggregated runtimes like Nu [218] and Zenix [96] provide another example, decoupling processes into fine-grained components, such as Nu's "proclets" or Zenix's serverless threads, with separate functions and data. These components can be offloaded to different servers to utilize their available CPU cores and memory, and they interact with each other via remote procedure calls (RPCs) to allow distributed application execution.

Through their primitives and abstractions, resource-disaggregated runtimes can capture rich program semantics (*e.g.*, data locality, access patterns, and shared data) and correspondingly apply runtime-level optimizations such as caching, prefetching, and locality-aware scheduling, thereby achieving higher efficiency than traditional DSM systems. However, they sacrifice transparency and require significant porting efforts, which may make them impractical for many legacy applications running in datacenters that were not originally designed with disaggregation in mind.

CHAPTER 3

Midas: A New OS Abstraction for Memory Harvesting

As a first step toward building a resource-efficient datacenter system stack, we focus on improving memory efficiency, which is critical since memory accounts for a substantial portion of datacenter operating costs. However, harvesting memory is challenging because the OS cannot easily reclaim or reassign memory once it has been granted and used for storing application data.

To tackle this problem, we revisit the first insight discussed in Section 1.2, which highlights that many datacenter applications possess elastic components that can utilize idle resources with unpredictable availability. In this chapter, we focus on application soft state—data that increases performance but is not required for correctness—as one concrete example of "elastic application components". To effectively manage and dynamically scale soft state, we introduce *soft memory*, an elastic virtual memory abstraction with *unmap-and-reconstrct* semantics that makes it possible for applications to use idle memory to store whatever soft state they choose while guaranteeing both safety and efficiency. We present Midas, a soft memory management system that implements this abstraction through an application-integrated runtime and a global coordinator within the OS kernel, achieving near-optimal application performance and memory efficiency.

3.1 Introduction

A wide range of applications can benefit from storing *soft state* in memory, including web applications [222], databases [168], key-value stores [163], CDN services [37, 176], and model serving frameworks [30]. Data is considered soft state when it is helpful for efficiency, but discarding it does not impact correctness because it can easily be reconstructed if it is later needed. For example, caches and memoization are both common forms of soft state. Soft state enables applications to trade extra memory consumption for better performance, and these gains generally increase with the amount of memory available [225, 241]. A significant fraction of memory is left idle in today's datacenters [150, 246], suggesting there is a large untapped opportunity to improve overall efficiency by using idle memory to store soft state.

While spending memory on soft state can improve performance, it must not compete with the need to store regular application data. For example, if too much memory is spent on soft state, this could lead to swapping to disk or worse still, out-of-memory errors, which can result in failures. Because of this, developers often limit their storage of soft state to a small static amount, for fear that they may run out of memory. In other words, it is a challenge to allocate enough soft state to consume all available idle memory, but to not go beyond the point where it would cause performance issues or failures.

Existing OS abstractions for elastically responding to changes in available idle memory are too limited. For example, the Linux Kernel maintains a *page cache* that automatically fills idle memory but it can only be used to cache disk blocks. This constrains idle memory to storing just a single type of soft state which may or may not provide the most utility for applications.

An ideal abstraction would instead democratize access to idle memory so that each application could choose how to best spend it (*i.e.*, the type of soft state that is most beneficial). For example, suppose an application does not rely much on local storage, but frequently accesses objects stored in a key value store over the network. Instead of being limited to the page cache, idle memory could be spent on caching the key-value store's objects locally, resulting in a much greater benefit.

This problem is further complicated in today's multi-tenant cloud. It is common for each server to run multiple applications, and they may come from different users and exhibit dramatically different performance sensitivity to the amount of soft state. At the same time, adding memory to one application can lead to reductions in the performance of others. Consequently, determining how to dynamically balance the soft state needs of different applications in a way that maximizes overall memory utility/performance is a challenge.

Insight. In this chapter, we aim to answer the following question: can we provide a new *virtual memory abstraction* for soft state (herein referred to as "soft memory") that developers can use to coordinate with the kernel so that they can take full advantage of all available memory? In other words, our goal is to no longer limit idle memory to the page cache, and to instead allow its use to be customized by each application in a way that maximizes overall utility.

Unlike existing systems that perform caching entirely in the user space [2, 30, 163], we propose *Midas*, a system that coordinates with the kernel to dynamically provision soft memory between applications. The advantage of this approach is two-fold. First, application developers can program with the illusion of an "unlimited cache", and are thus freed of the burden of manually managing their soft state. To avoid running out of memory, the kernel responds to memory pressure by rapidly *unmapping* soft memory pages. To transparently recover any lost soft state, later accesses will automatically trigger the application to *reconstruct* the missing soft state. Second, the kernel has global visibility of all applications, their memory usage, and the amount of idle memory, making it possible to understand each application's sensitivity to memory size and automatically coordinate soft memory allocation between applications. Midas also incorporates the page cache by treating it as another source of soft memory.

Challenges. Midas is a soft memory management system that achieves (1) programming flexibility and (2) dynamic memory provisioning, with *unmap-and-reconstruct* semantics, to guarantee both safety and efficiency. Realizing these benefits requires overcoming four major challenges:

First, what interfaces shall we expose to developers? To improve usability, Midas provides developers with a *soft memory pointer* abstraction (similar to C++ smart pointers) to access soft memory easily and safely (see §3.4.1). Midas offers a set of high-level key-value store APIs, which are similar to those of popular cache services (such as Memcached [163], Redis [2], CacheLib [30], *etc.*), but enhanced to allow the exposure of more semantics to the runtime. A critical interface we expose to developers is *data structure reconstruction*—developers not only register soft memory objects but also specify their (re)construction logic, so soft state can be transparently regenerated if it is later accessed after it was evicted.

Second, how shall soft memory be managed? Program data is allocated as objects on the heap but the kernel cannot recognize them, as it is only aware of memory pages. As a result, if we let the kernel manage soft memory alone, it could only reclaim space in coarse-grained units without knowledge of what objects the space contains. For example, reclaiming hot (*i.e.*, frequently accessed) objects in a soft-state cache can lead to significant slowdowns. In addition, it is undesirable to reclaim space from the programs that would benefit the most from soft state when others need it less, but such performance sensitivity information is invisible to the kernel.

To solve the problem, we propose a runtime library that can be linked into each application to recognize object behaviors, letting the runtime and the kernel *co-manage* soft memory. The Midas runtime offers a log-structured allocator [220] and a concurrent evacuator that continuously identifies and compacts hot objects into a small soft memory space. This information (of hot and cold regions) is shared with the kernel so that it can focus its reclamation on regions with cold objects (see §3.4.2).

Third, how can we coordinate soft memory allocation between applications? The runtime can only see each application's individual behavior without any global knowledge of the server's available memory and other applications' needs. Furthermore, the runtime can only manage objects in user space, but *cannot* dynamically add/remove memory between applications. To overcome this challenge, we propose a global coordinator inside the Linux kernel. The coordinator periodically probes each application by communicating with the runtime to request information regarding the application's sensitivity to cache size. Cold regions of soft memory from applications that are less sensitive to size changes will be reclaimed and memory will be given to those that are more sensitive (and hence benefit more from a larger cache) by the kernel (see §3.4.3).

Finally, how can the kernel quickly reclaim soft memory without disrupting a running application? Since the kernel operates at page granularity, a natural idea is to swap out pages that contain soft-state data. Unfortunately, swapping is disruptive—swapping out a page blocks all incoming memory allocations and hence all threads of the application; frequently swapping pages can introduce significant overheads that prevent applications from reaching service-level agreements (SLA) [216] (see §3.2).

To maintain high efficiency, Midas instead uses the kernel to unmap pages directly (which is much faster than swapping them to disk). When pages are unmapped, their underlying data is lost—this is acceptable for soft state because it can be regenerated. Without coordination, however, the kernel cannot distinguish soft state from application data, making unmapping potentially unsafe.

To solve this problem, our runtime is designed in a way that is resilient to data loss. A soft pointer-based interface detects data loss through segmentation faults that are triggered by the runtime's functions. These functions are carefully designed to capture faults and transparently invoke a *reconstruction* interface to regenerate the needed data (see §3.4.2.3).

Compared to paging, Midas does not freeze the execution when shrinking soft memory, resulting in less disruption to the application. Furthermore, reconstruction focuses on recovering the individual objects that are needed and hence is much more fine-grained and can be more efficient than swapping, which brings back entire pages. Reconstruction may require more computation than paging (the amount of computation depends on exact soft state data). Therefore, Midas provides a profiling tool that warns developers when reconstruction incurs a high cost (discussed in §3.4.4).

Results. With Midas, one can easily allow applications to take advantage of soft state that do not currently support it. It is also easy to port legacy code that uses an existing cache system to use Midas instead. Our evaluation shows that Midas can efficiently and safely harvest idle memory to store applications' soft state and achieve near-optimal performance while reacting to extreme memory pressure quickly enough to avoid running out of memory. By effectively granting soft memory to the applications that benefit the most, Midas achieves 1.34× higher overall throughput than Cliffhanger (a state-of-the-art caching system). Midas is available at `https://github.com/uclasystem/midas`.

3.2 Motivation

Many types of applications can benefit from soft state. For example, a web frontend could cache content locally after loading it from a backend to reduce network traffic and improve response times; a database could cache the results of user queries to reduce disk I/O and improve throughput; and a data analytic or machine learning system could memoize intermediate computation results to eliminate redundant computations.

To gain a high-level understanding of how much improvement can be achieved by storing soft state, we experimented with three datacenter applications: SocialNet (from DeathStar-Bench [80]), MongoDB [168], and HDSearch (from µSuite [236]). Each of these applications are capable of using soft state. SocialNet [80] is a web forum built using microservices; it employs Memcached and Redis to cache user data in its frontend services. MongoDB [168] is a NoSQL database; it has a built-in, in-memory caching engine that caches recently queried data. HDSearch is an image search service that memoizes the feature vectors of the images in its corpus, generated by a GPU-based DNN.

Figure 3.1 shows the throughput of each application with varying amounts of soft state. The *x*-axis represents the percentage of each application's working set cached in memory, and the *y*-axis shows the normalized throughput (to its performance without soft state). Soft

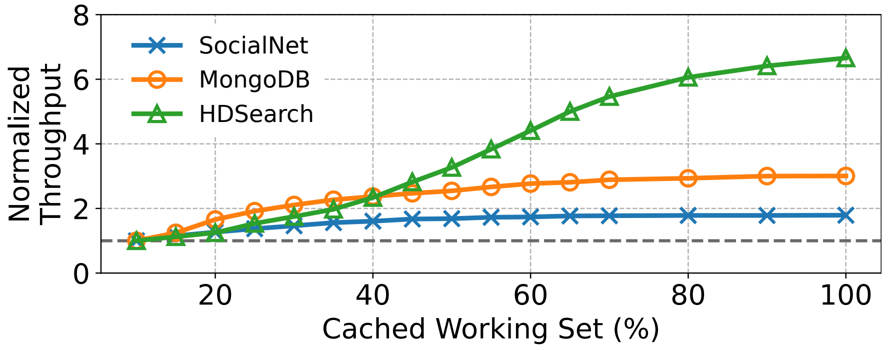


Figure 3.1: The throughput of all three applications increases by caching more soft state, but the benefit varies: SocialNet is 1.8× faster by caching 70% of its working set, while HDSearch, in contrast, achieves a 3.3× throughput increase by caching 50% state.

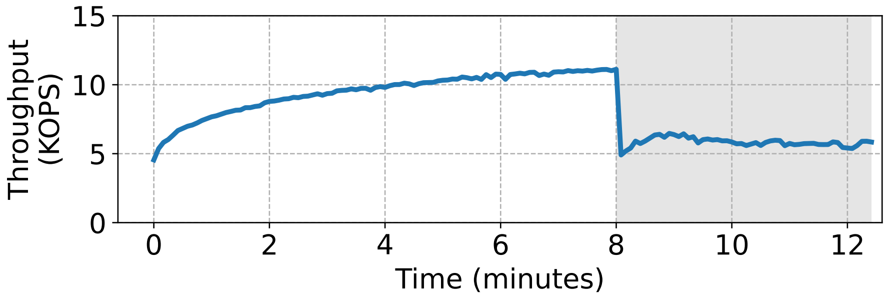


Figure 3.2: SocialNet starts to swap when it caches excessive data and exhausts all available memory at t = 8min and it experiences a throughput collapse.

state is helpful to all applications but the amount of benefit it provides varies. SocialNet is the least sensitive to its soft state size; however, it still sees a 1.8× speedup by storing 70% of its soft state. HDSearch, in contrast, is more sensitive to the soft state size—its throughput increases by more than 3× with only 50% of its soft state.

Real-world datacenter applications can access a massive amount of data. For example, a web forum like Twitter generates petabytes of new data every day [249]. Thus, blindly storing soft state in memory without a proper limit can hurt application performance. An example of this problem is shown in Figure 3.2. Storing soft state increases the throughput of SocialNet up to a point. However, when idle memory becomes exhausted, the kernel begins

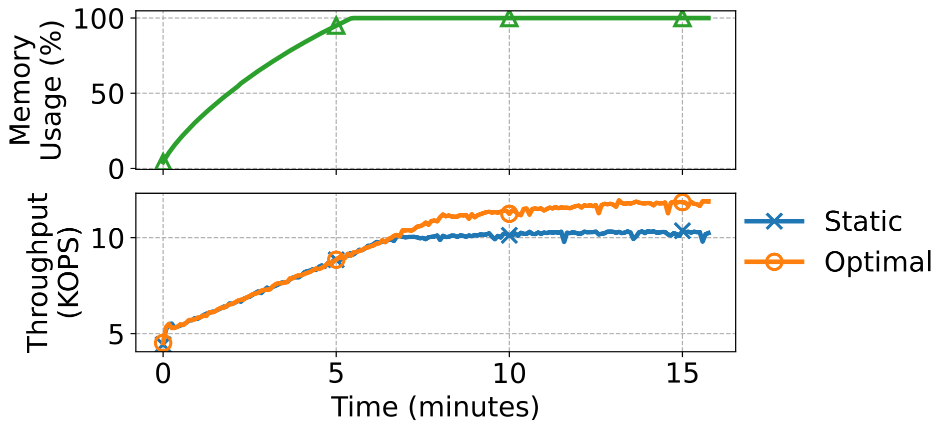


Figure 3.3: Statically provisioning the cache space for SocialNet is suboptimal. During t =0min–5min, the cache is overprovisioned which wastes memory. After that, the cache becomes underprovisioned which limits performance.

to swap out pages (at t = 8min), leading to a severe collapse in throughput.

A simple strawman solution is to statically provision a limited memory capacity for storing soft state so that memory use does not grow unbounded. However, provisioning the right capacity is extremely challenging in practice.

First, for each application, we must find its sweet spot of cache capacity; underprovisioning limits performance while overprovisioning wastes memory. In addition, datacenter applications often have phased behaviors and load variability [19, 22], making it impossible to have a simple static configuration that is optimal at all times. For example, Figure 3.3 shows the results of SocialNet when statically provisioning it with 4 GiB for storing soft state. It takes about 5 minutes to fully fill this memory, leading to a suboptimal utilization during this period. Performance increases with more usage until it exhausts the soft state limit. After that, performance flattens out despite the possibility of higher throughput if additional soft state memory were available (the optimal line).

Second, as shown by Figure 3.1, different applications gain different amounts of benefits through caching. To achieve optimal overall performance with a limited amount of memory, one must grant space correctly to the applications that benefit the most. For example, initially

MongoDB's performance is most sensitive to the amount of soft state (the left side of Figure 3.1), and thus we should prioritize its need. However, the return diminishes quickly after caching 30% of its state. To make the best use of the remaining memory, we should respond by granting memory to HDSearch.

These problems call for a new system that can provide *elastic* access to soft state for applications and *dynamically coordinate* usage among applications in response to each one's execution phase and sensitivity to soft state size. To be efficient, soft state should be able to quickly scale up and down its capacity with little disruption. To be safe, the system should be resilient to data loss caused by scaling down. To be responsive, the system should conduct coordination among applications quickly. Finally, to be practical, the system should provide familiar programming abstractions for developers to store and access soft state.

3.3 Midas Overview

As shown by Figure 3.4, Midas consists of three main components: a programming abstraction for using soft memory (§3.4.1), an application-integrated runtime that manages soft-memory objects (§3.4.2), and a global coordinator that arbitrates soft memory usage across different applications (§3.4.3).

Midas provides programming abstractions that enable simple and efficient use of soft memory through familiar APIs. At a low-level, programmers can interact with Midas through *soft memory pointers*, an abstraction that provides object ownership similar to C++ smart pointers. However, it differs in that underlying objects can be forcibly released when under memory pressure, even if still in scope. If a released object is later accessed, a *reconstructor* function is invoked to regenerate the missing object (*e.g.*, by fetching it from a database over the network).

Building upon soft memory pointers, Midas provides a higher-level library of familiar STL-style *soft data structures*—including arrays, hash tables, and queues. These hide

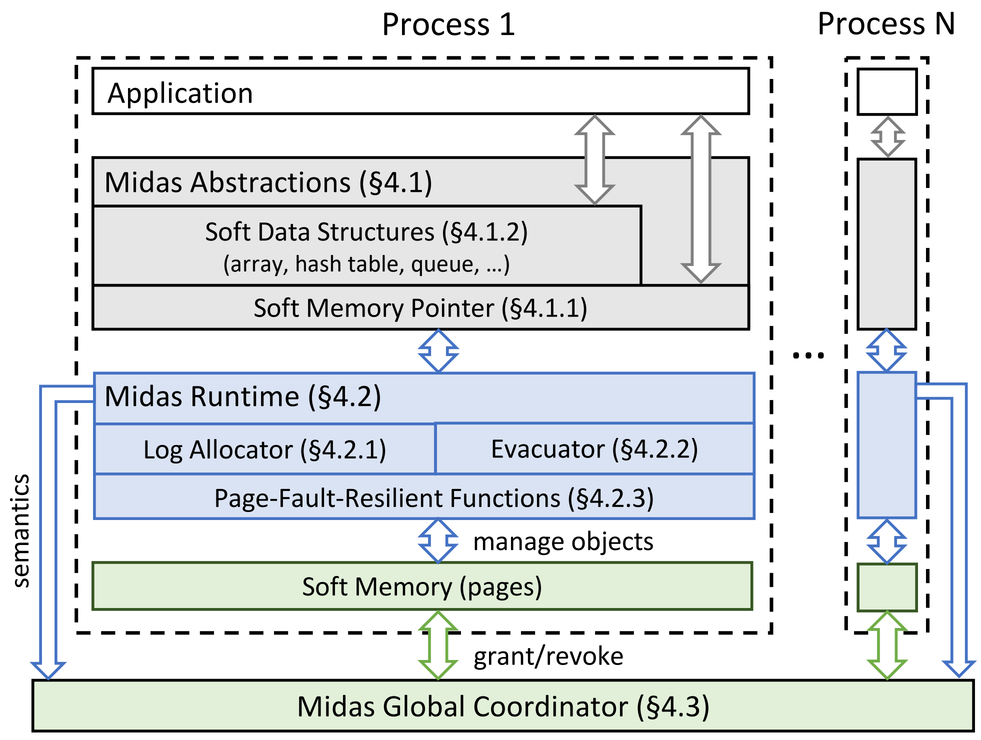

Process 1
Process N
Application
Midas Abstractions (§4.1)
Soft Data Structures (§4.1.2)
(array, hash table, queue, …)
Soft Memory Pointer (§4.1.1)
Midas Runtime (§4.2)
Log Allocator (§4.2.1)
Evacuator (§4.2.2)
Page-Fault-Resilient Functions (§4.2.3)
manage objects
Soft Memory (pages)
grant/revoke
semantics
…
Midas Global Coordinator (§4.3)

Figure 3.4: Midas enables developers to utilize soft memory easily and efficiently with three major components: a familiar programming abstraction, an application-integrated runtime, and a global soft-memory budget coordinator.

the complexity of managing individual soft memory pointers, and can be used as drop-in replacements for existing data structures. For example, a developer building a key-value store similar to Memcached could use a soft hash table to store soft memory objects. Midas's high-level interface is generally sufficient for most use cases, but developers are free to build their own custom soft data structures through use of soft memory pointers.

Midas manages soft memory objects through a runtime that is linked as a library with the application. It serves as an allocator for soft memory objects. It works cooperatively with the coordinator (discussed next) to determine the best memory to release (*i.e.*, idle memory first, then cold objects, and finally hot objects). To achieve this, the runtime provides a moving allocator that embraces the idea of log-structured memory [220] to organize soft memory into different segments. An evacuator thread scans and compacts logs to segregate hot objects, cold objects, and dead objects. This helps both to coordinate which memory should be freed and to reduce fragmentation.

However, the runtime is not trusted for correct operation. If it fails to respond quickly enough or if memory pressure becomes too severe, pages will be unmapped in an uncoordinated fashion to avoid swapping. In the event such forcible revocation happens, the runtime is designed to safely tolerate page faults when accessing unmapped memory. To achieve this, we developed a set of page-fault-resilient functions and used them as primitives to build our runtime.

Midas's global coordinator dynamically adjusts the soft memory budget among applications to optimize their overall performance. It periodically probes the marginal utility of soft memory for each application by granting a small amount of additional memory and observing the effect on performance. Using this information, the coordinator can optimize the allocation of soft memory by granting it to the applications that benefit the most. The coordinator defines the global utility function as the weighted average of all applications' performance and employs a hill-climbing algorithm to approach the global optimal point. Midas allows operators to specify the weight of each application to indicate relative significance, similar to the `nice` interface of Linux.

3.4 Design

3.4.1 Soft Memory Abstraction

Soft memory is a new type of memory that can be revoked under memory pressure. In Midas, soft memory is backed by physical pages that can be unilaterally unmapped and reclaimed by the OS kernel. Accessing reclaimed soft memory will trigger a *reconstruction event* to rebuild the missing data. Midas provides a smart-pointer-like API to enable developers to easily use soft memory, hiding the complex details of soft memory allocation/deallocation,

```cpp
template <typename T, typename... ReconArgs>
class SoftMemPool {
  SoftMemPool(std::function<T(ReconArgs...)> reconstructor);
  SoftUniquePtr<T, ReconArgs...> new_unique();
  SoftSharedPtr<T, ReconArgs...> new_shared();
};

template <typename T, typename... ReconArgs>
class SoftUniquePtr {
  ~SoftUniquePtr();
  T read(ReconArgs... args);
  void write(T newval);
  bool cmpxchg(const T &oldval, T newval);
};
```

Listing 3.1: Midas's soft memory pool and unique pointer interface.

page-fault handling, and data reconstruction (§3.4.1.1). Furthermore, Midas offers high-level data structure libraries as composable building blocks (§3.4.1.2).

3.4.1.1 Soft Memory Pointer

Listing 3.1 shows Midas's soft memory pool and pointer interface. To use soft memory, developers first need to create a soft memory pool which can later be used to allocate soft memory pointers. The pool abstraction conceptually groups together soft pointers whose objects can be reconstructed in a similar way. Midas exposes the pool as a C++ template class whose parameters consist of two parts: `T`, which is the object type of soft pointers to allocate, and `ReconArgs`, which are the types of arguments used for reconstructing a missing object. Developers can initialize a pool with a `reconstructor` function and then allocate pointers using `new_unique` (for soft unique pointers, similar to C++'s `std::unique_ptr`) and `new_shared` (for shared pointers).

Soft memory pointers support automatic lifetime management through reference counting. Developers can use its `read` API to get the value of the pointed object. In case the underlying soft memory has been reclaimed, Midas will automatically reconstruct the missing object using the reconstruction arguments passed into `read` (we will show a concrete example soon in §3.4.1.2). Midas hides the raw reference and returns the value by *copying*. This is critical as the underlying reference may become invalid any time when the soft memory gets reclaimed. With copying, Midas restricts potential faulting sites to stay inside Midas's internal code, thereby freeing developers from handling complicated page faults in the application code. The copying design incurs negligible performance overheads (only a few additional cache accesses). `write` enables developers to update the object value. However, different from `read`, `write` does not require reconstruction arguments as Midas can directly rebuild the object using the new value. Soft pointers also support atomic operations like compare-and-exchange, enabling developers to atomically update object values to support multi-threaded applications.

With its smart pointer design, Midas is able to capture rich application semantics for effectively managing soft memory. For example, since all soft object accesses go through the `read/write` API, Midas can accurately track the hotness information of each object which can be leveraged by Midas runtime for making intelligent object placement and eviction decisions (details in §3.4.2). Soft pointer's automatic lifetime management enables cascading eviction, improving the efficiency of using soft memory. For instance, in a web forum application, a forum post object may contain a soft unique pointer to an attached picture. Under memory pressure, Midas may decide to evict the post object in which case the reference count of the picture pointer will automatically drop to zero and trigger evicting the dangling picture object cascadingly.

```
1  template <typename T> class SoftArray {
2    SoftArray(size_t size, std::function<T(size_t)> reconstructor);
3    T read(size_t idx);
4    void write(size_t idx, T t);
5    bool cmpxchg(size_t idx, const T &oldval, T newval);
6  };
7
8  class BlockCache {
9    BlockCache(size_t sz) : array_(sz, [](size_t idx) {
10     return read_from_storage(idx); }) {}
11   Block read(size_t idx) { return array_.read(idx); }
12   void write(size_t idx, Block block) {
13     array_.write(idx, block);
14     write_to_storage(idx, block);
15   }
16   SoftArray<Block> array_;
17 };
```

Listing 3.2: Midas's soft array interface and a simple user-level storage block cache (similar to Linux's page cache) built using soft array.

3.4.1.2 Soft Data Structures

To further reduce the programming effort of using soft memory, Midas offers high-level data structures as convenient building blocks. Midas's built-in data structures include soft arrays, soft hash tables, and soft queues; developers can also easily build more based on the soft pointer abstraction.

Listing 3.2 presents the interface of soft array (lines 1-6). Developers can create a soft array by specifying its size and reconstructor (which rebuilds the array element of a given index). Soft array supports standard read, write, and atomic operations by index. Under the hood, a soft array is simply implemented via an ordinary array of soft pointers.

Lines 8-17 present a user-level storage block cache as a simple illustrative application, similar to Linux's page cache. `BlockCache` internally wraps a soft array whose elements are storage blocks (line 16). This enables it to efficiently leverage idle memory to cache storage blocks in a best-effort manner. For each block read request, it simply retrieves the result from the soft array (line 11). Upon an element miss, the array automatically reconstructs the element by reading the block back from the storage device (lines 13-14). For each block write request, `BlockCache` updates both the cache in array and the data in storage.

3.4.2 Application-Integrated Runtime

Midas runtime is the key component that manages soft objects to enable efficient use of soft memory. It includes a log-structured memory allocator that serves memory allocation requests and organizes objects into a list of segments (§3.4.2.1), a concurrent evacuator that constantly compacts hot objects and releases cold and dead objects (§3.4.2.2). Page faults can happen in Midas runtime when the soft memory it is accessing gets reclaimed and unmapped because of memory pressure. To ensure robustness, we carefully built the runtime using a set of page-fault-resilient functions which are able to capture page faults and gracefully recover from them (§3.4.2.3).

In Midas, the runtime as well as the soft memory it manages are linked directly into each application's address space. Compared to traditional cache services (*e.g.*, Memcached) that run in a separate process, our design offers several important advantages. First, it provides direct and efficient soft memory accesses for applications, eliminating the inter-process communication (IPC) overhead. Second, it enables our runtime to profile the application and collect semantics, greatly facilitating semantics-aware optimizations. Third, since each application has its own runtime, we can easily enforce soft memory isolation among applications and adaptively customize the memory management policy of each application.

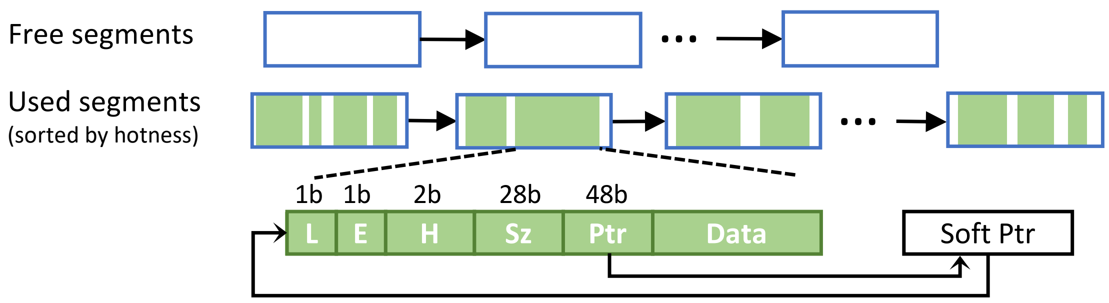


Figure 3.5: Midas organizes soft memory using a free segment list and a used segment list (sorted by segment's hotness, useful for Midas's evacuator in §3.4.2.2). It employs a log-structured allocator to serve memory allocation requests. Each object has a 10-byte header, which includes a liveness bit, an evacuating bit, hotness bits, an object size field, and a reversed pointer field.

3.4.2.1 Log-structured Soft Memory Allocator

Midas embraces the idea of log-structured memory [220] to manage soft memory; it reduces memory fragmentation through compaction, thereby achieving higher efficiency in utilizing soft memory.

Midas's log-structured allocator organizes soft memory using a free segment list and a used segment list, illustrated in Figure 3.5. Segments are the units for Midas to perform evacuation to compact objects and reclaim space (details in §3.4.2.2). The total size of all segments (used and free) equals the soft memory budget that the linked application receives from the global coordinator (§3.4.3). For each memory allocation request, the allocator allocates space from a free segment; if the current one is full, it will pop a new one from the free list. Midas backs each segment using a 2 MiB huge page; this reduces TLB pressure and page table walk cost. While small objects reside in only one segment (i.e., they do not cross the segment boundary), big objects whose sizes are larger than 2 MiB span across multiple segments. Since the free list does not provide any address contiguity guarantee for segments, Midas breaks the big object into smaller pieces—each one fits into a single segment—and chains them together using segment headers. The decomposition is transparent to application developers; upon object read, Midas automatically reads all segregated pieces and stitches them back. This is possible thanks to Midas's pass-by-copy interface (§3.4.1).

Each allocated object has a 10-byte header inlined with its data, used for tracking the object's runtime information. This includes 1) a liveness bit, indicating whether the object has been deallocated; 2) an evacuating bit, marked by the evacuator to synchronize evacuation with object accesses; 3) hotness bits, a counter that will be incremented (or unchanged when it has reached the maximum) each time the object gets accessed; 4) a size field, indicating the total size of the object; 5) a reverse pointer field, used by the evacuator, if it moves the object, to rewrite the soft pointer.

3.4.2.2 Soft Memory Evacuator

As the allocation goes on, the application may eventually deplete the free segment list. It is the responsibility of Midas's evacuator to constantly release cold and dead objects, ensuring the best use of soft memory by only storing hot objects. In addition, the evacuator tracks segments in order of hotness in a used list (see Figure 3.5), to simplify the design and improve the speed of memory reclamation, in which the kernel forcibly unmaps application's soft memory pages under intense memory pressure (§3.4.3).

Midas's background thread continuously monitors the free segment ratio and triggers evacuation if it falls below a configurable threshold (our default value is 90%). The evacuation mainly consists of three stages:

Scanning Stage. The evacuator first scans through all objects in the used segment list. For each scanned object, it decrements the embedded hotness counter (similar to the CLOCK algorithm [57]). The evacuator treats objects with a zero pre-scanning hotness value, in addition to deallocated objects, as *dead* objects; they will be released in the compaction stage. The evacuator calculates the *live ratio* of each segment (*i.e.*, the percentage of live bytes) during scanning, and then uses it to sort all scanned segments to decide their priority for compaction. The segment with the lowest live ratio will be compacted first as it yields the largest benefits (in terms of the reclaimed space).

Compaction Stage. The evacuator compacts one segment at a time. For each live object, it first relies on the evacuation bit to synchronize with application threads to avoid data race (similar to AIFM [216]). It then copies the object into a new segment and leverages the reversed pointer field to rewrite the address of the corresponding soft pointer. After evacuating all live objects, it moves the segment from the used list into the free list.

Sorting Stage. After compaction, the evacuator calculates the segment-level hotness value for all segments in the used list, defined as $\sum_{\forall obj \in seg} SIZE(obj) \cdot HOTNESS(obj)$. It finally sorts the used list by segment-level hotness in ascending order.

3.4.2.3 Page-Fault-Resilient Functions

Midas runtime directly manipulates soft memory during allocation and evacuation. Since the kernel may unmap soft pages to reclaim memory under pressure (details in §3.4.3), the runtime has to be aware of page faults and be able to recover from them gracefully. We carefully built the runtime to achieve this goal. First, we stored the important metadata (e.g., the free and used segment lists) in normal memory instead of soft memory, therefore it will not be lost under memory pressure. This is viable as the metadata only consumes little memory (less than 10 MiB). Second, we introduced *page-fault-resilient functions* and used them as primitives to build the runtime.

A page-fault-resilient function is able to capture any internal page fault that stems from dereferencing unmapped soft memory and respond to it by reverting all side effects and throwing a `SoftMemUnmapped(fault_addr)` exception to the caller. As a concrete example, in Midas we internally implemented a page-fault-resilient memory copy function, which is

```
1  for each segment S to compact {
2    D = pick_destination_segment();
3    for each object O in S {
4      try {
5        // A wrapper around our PF-resilient memcpy
6        copy_object_into(O, D);
7      } catch (SoftMemUnmapped &exception) {
8        if (exception.fault_addr belongs to O)
9          break; // Skip S as it has gone
10       else // It must belong to D
11         goto line2; // Pick a new D and restart
12     }
13   }
14 }
```

Listing 3.3: Midas implements its evacuator's compaction code using a page-fault-resilient memory copy function.

used to build the evacuator's compaction code to withstand page faults (see Listing 3.3). Page faults can happen when copying objects from the old segment into the new segment. To deal with this case, Midas uses its resilient memory copy function (line 6) to capture and handle the potential exception (lines 7-12).

Midas registers its own signal handler to facilitate capturing and handling all soft-memory-related page faults. Additionally, a page-fault-resilient function satisfies the following requirements to ensure resilience:

- It embeds a fault recovery code block for aborting the partial execution and rolling back side effects. Midas runtime maintains a mapping from resilient functions to their recovery blocks so that when page fault happens the handler can invoke the corresponding recovery code.

- All of its inner non-resilient functions have to be inlined to prevent the control flow from jumping out of its scope. Otherwise, the page fault handler is unable to find the corresponding recovery code.
- It preserves its stack frame base pointer (by disabling the compiler optimization) so that the fault handler can easily unwind its stack and throw an exception back to its caller.

3.4.3 Global Soft Memory Coordinator

Midas's global coordinator is responsible for granting server's idle memory to applications as soft memory and coordinating the budget across applications to optimize the overall performance.

3.4.3.1 Soft Memory Management Mechanism

The coordinator maps idle memory pages directly into an application's address space as soft memory segments. For each application, the coordinator dynamically maps or unmaps pages to readjust its soft memory budget. To facilitate the management, the application's runtime shares its free segment list and used segment list with the coordinator.

To grant more soft memory to an application, the coordinator maps more pages to it and inserts them into the free segment list. Similarly, to reclaim memory from an application, the coordinator unmaps pages. The coordinator first tries to pop out and unmap the segments from the free list; since they do not hold any useful live objects, unmapping them does not incur any impact on the application's performance. Meanwhile, the runtime strives to avoid the exhaustion of the free list by triggering evacuation (§3.4.2.2).

The synergy between the runtime and the coordinator is able to handle moderate memory pressure (*i.e.*, the common case). However, under severe pressure, the evacuation may fall behind, leading to an empty free segment list. To avoid depletion, the coordinator reacts by unmapping used segments which may induce performance disruption in two folds. First, when the application later tries to access an unmapped object, the runtime will experience a page fault which incurs overhead. Second, the runtime has to spend additional time reconstructing the missing object. To alleviate this issue, the coordinator prioritizes cold segments over hot segments. Thanks to the evacuator, the segments in the used list have been ordered by their hotness (§3.4.2.2). Therefore, the coordinator can realize prioritization by simply unmapping segments based on their order in the list.

3.4.3.2 Coordination Policy

Midas continuously adjusts each application's soft memory budget by solving the following optimization problem:

$$\underset{m}{\text{maximize}} \sum_{\forall i \in APPS} w_i \Gamma_i(m_i) \text{ , subject to } \sum_{\forall i \in APPS} m_i = M$$

For each application i, w_i denotes its weight (which is either specified by the operator or uses the default value 1) and Γ_i denotes its performance utility when assigned soft memory of size m_i. The server-wide overall utility is defined as the weighted sum of all application's utilities. M denotes the server's total idle memory.

By default, the coordinator estimates Γ_i as $-RCOST_i$, where $RCOST_i$ is the application's CPU usage spent on reconstructing missing objects. Midas's runtime can easily collect this per-application information and report it to the coordinator. Developers can also plug in the real performance metric reported by applications—which already exists in many datacenter applications [33]—for a more faithful Γ_i.

Midas solves the optimization problem using the hill climbing approach [221]. It periodically probes every application's marginal utility benefit $\frac{\partial \Gamma_i(m_i)}{\partial m_i}$ by additionally assigning a small portion of memory Δ_{m_i} and monitoring the change of utility Δ_{Γ_i}. Midas regrants the soft memory budget from the application with the lowest marginal utility benefit to the one with the highest benefit.

In contrast, Cliffhanger (a recent cache service) [52] adopts a coordination policy that optimizes for the overall cache hit rate, but this does not necessarily optimize the overall performance. For example, caching objects that are frequently accessed may not be helpful if they can be cheaply reconstructed. Midas avoids this issue by using both access frequency and reconstruction cost as metrics for optimization.

3.4.4 Discussion

Though Midas is mainly designed for caching hot data and memoizing intermediate computation results, developers have the freedom to put any data into soft memory as long as it is reconstructible. However, storing data that is expensive to reconstruct but infrequently accessed can lead to performance issues. Midas provides a profiling tool that generates runtime warnings if such cases are detected. In addition, Midas offers a debugging mode where we validate the reconstruction logic by calling the user-defined reconstruction function and comparing its result with the actual cached object using the object's comparison operator. Bugs are reported if these objects are not identical.

Midas also incorporates Linux's page cache by simply treating it as another per-application soft memory pool. For each application, Midas's shim layer intercepts all POSIX file operations and caches the file data using a soft hash table, whose keys are file inode numbers along with block-aligned offsets and values are file blocks. The reconstructor rebuilds the missing block by performing the actual file read.

3.5 Implementation

Midas is implemented in C++ and includes bindings for C. Our implementation has 2,814 LOC for the soft memory abstraction (§3.4.1), 3,866 LOC for the runtime (§3.4.2), and 1,029 LOC for the global coordinator (§3.4.3).

Soft data structures store their metadata (*e.g.*, a hash table's bucket array that stores indices) in normal memory and store their data payload (*e.g.*, a hash table's key-value pairs) in soft memory using soft pointers.

The log-structure allocator enforces 16-byte alignment for allocated data to make it GCC-compatible. The evacuator adopts a concurrent pauseless design similar to AIFM [216]. The evacuator ensures atomicity when evacuating or reconstructing large objects that span across multiple soft memory segments. Midas registers its own `SIGSEGV` handler. For each segmentation fault, the handler checks whether the faulting memory address belongs to a soft memory region and whether the faulting program counter (PC) belongs to a page-fault-resilient function; for faults that do not meet these conditions, the handler treats them as unrecoverable exceptions and aborts the program. To facilitate the PC check, Midas leverages a linker script to place all resilient functions into a separate code segment whose layout is known at compile time.

During each application's initialization, the runtime registers itself to the global coordinator using `ioctl` and uses `mmap` to create a shared memory region for exposing information—including its free segment list and used list (implemented as arrays) and the application's reconstruction cost (implemented as a counter)—to the coordinator.

We implemented the global coordinator as a user-space daemon (that runs the coordination policy) and a privileged kernel module (that executes the coordination decision by mapping/unmapping pages to/from user processes directly). Every 5 seconds, the coordinator probes the marginal utility of each application and makes a new adjustment to soft memory budgets. It probes an application by either granting or revoking 64 MiB soft memory and monitoring its performance change. In each adjustment, it regrants up to 256 MiB soft memory from the application with the lowest marginal utility to the one with the highest utility. To avoid oscillation, it refrains from granting more soft memory to the application until it has consumed the additional memory offered in the previous round.

Applications	**Abstractions used**	**Porting effort (LOC)**	**CPU cores**	**Normal mem. (GiB)**	**Peak soft mem. (GiB)**	**Reconstruction cost (µs/obj.)**	**Dataset**
HDSearch [236]	Soft hash table	36	12	1.7	13.6	1244.2	`OpenImg` [127], 1.9M images
WiredTiger [169]	Soft pointer	332	12	3.7	21.3	20.6	Facebook `USR` [22], 50M KV
Storage Server [124]	Soft array	29	4	1.1	20.4	10.5	`multilate` [108], 16 GiB disk
SocialNet [80]	Soft hash table Soft queue	175	20	1.3	12.2	99.1–3227.7	Socfb-Penn94 [214], 41.5K nodes, 1.4M edges

Table 3.1: We ported four applications into Midas with low programming effort. All four applications extensively use soft memory while their data reconstruction costs vary drastically.

3.6 Programming with Midas

We present general guidelines of programming with Midas (§3.6.1) followed by concrete examples of porting four real applications (§3.6.2).

3.6.1 Guidelines

When is it safe to use soft memory? Developers can generally use soft memory to store any application data that follows the unmap-and-reconstruct semantics. To support evacuation, developers have to implement copy constructors for objects stored in soft memory.

When is it beneficial to use soft memory? Developers should generally consider using soft memory when applications can opportunistically benefit from having additional memory. Typical use cases include caching in web applications and memoization in data analytics systems. They often have unknown marginal utility and unbounded memory footprint, making them hard to handle efficiently through static provisioning. Midas can benefit them by automatically rightsizing their soft memory budget and harvesting idle memory.

How to migrate from traditional cache services? Existing applications that employ local cache services (*e.g.*, Memcached [163] or Redis [2]) can directly use Midas as a drop-in replacement. Existing applications that employ distributed cache services (*e.g.*, AWS ElastiCache [1]) can use Midas as a fast local cache tier to reduce the overhead of accessing remote cache.

3.6.2 Application Case Studies

We ported four applications to Midas. They cover a range of CPU usage, normal and soft memory usage, data reconstruction cost, and Midas's abstraction usage (see Table 3.1).

HDSearch [236] is an image search service based on content similarity. For each query, a feature extraction backend transforms the input image into a feature vector via a DNN (running on GPU), and then caches the result along with a hash of the image (using Memcached for memoization). To port this application, we replaced Memcached with our soft hash table, which only involves 36 LOC changes. It has 1.7 GiB normal memory usage and 13.6 GiB peak soft memory usage. Reconstructing KV pairs is expensive (1244.2 µs per object) as it requires re-performing transformation on GPU.

WiredTiger [169] is a NoSQL key-value storage engine used by MongoDB [168]. It persists all key-value pairs in storage indexed via an in-memory B+ tree. It has a built-in in-memory caching engine that caches the data of B+ tree's internal nodes and leaf nodes to reduce expensive storage I/Os. To port WiredTiger, we implement its caching engine using Midas's soft memory pool and pointer abstractions; we created a soft memory pool with a reconstruction method that wraps WiredTiger's existing code for handling cache misses, and replaced ordinary B+ tree pointers with soft memory pointers allocated from the pool. This only involves 332 LOC changes. With our port, WiredTiger has 3.7 GiB normal memory usage and 21.3 GiB peak soft memory usage. Reconstructing a tree node object requires reading its content from the disk and rebuilding the index, which takes 20.6 µs.

Storage Service is an NVMe-based block storage service similar to Reflex [124]. It exposes a standard block I/O interface using RPC to support accessing 4KiB storage blocks remotely. Its original design uses SPDK [3] to communicate with the storage block device, which bypasses Linux's page cache. To port it, we cache the block data using a soft array, similar to the `BlockCache` design in Listing 3.2. This requires adding 29 LOC. With our port, it uses 1.1 GiB normal memory and 20.4 GiB peak soft memory. Reconstructing an array element requires a block I/O which takes 8.5 µs to finish.

SocialNet is a twitter-like latency-critical web service from DeathStarBench [80]. It is built using 12 microservices with sophisticated fan-out patterns and call dependencies. Its original design uses Memcached/Redis to cache users' data and memoize results of certain queries, and employs pools to cache TCP connections/RPC sessions. Since each microservice has its own binary and runs within its own process, Midas treats SocialNet as 12 different applications. To port it, for each microservice, we replace its Memcached/Redis usage with Midas's soft hash table and connection pool with Midas's soft queue; this involves 175 LOC changes. With our port, it uses 1.3 GiB normal memory and 12.2 GiB peak soft memory. It takes 99.1–3227.7 µs to reconstruct an object depending on its type; for example, it takes only 99.1 µs to re-establish an RPC session but requires 3227.7 µs to re-fetch a user's post.

3.7 Evaluation

Setup. We conducted experiments on one server that equips a 48-core Intel Xeon Gold 6252 CPU and 128 GiB memory. The server ran Ubuntu 20.04 with Linux 5.14. In line with prior work [207], we enable hyperthreading, but disable dynamic CPU frequency scaling, transparent huge pages, and kernel mitigations for transient execution attacks. For interactive services (*e.g.*, SocialNet), we use a separate server to generate load, which connects to the application server via a 10 GbE network. For all four applications, we generated requests with Zipfian distribution, consistent with the study of real datacenter workloads [30].

Our evaluation seeks to answer the following questions:

1. Can Midas judiciously coordinate soft memory among applications to optimize overall performance? (§3.7.1)

2. Can Midas quickly and reactively harvest available idle memory to improve utilization and performance? (§3.7.2)
3. Can Midas quickly react to memory pressure to avoid out-of-memory killing while maintaining good performance? (§3.7.3 and §3.7.4)
4. How does the data reconstruction cost of an application affect its performance? (§3.7.4)

3.7.1 Coordinating Soft Memory

In this experiment, we investigated whether Midas can judiciously coordinate soft memory usage among applications to optimize overall performance.

We provisioned the server with 20 GiB idle memory and co-ran all four applications (§3.6) using Midas. Initially, all applications start with the same amount of soft memory (*i.e.*, 5 GiB), but Midas will dynamically adjust it. SocialNet has 12 loosely-coupled microservices and we start by evenly splitting the 5 GiB budget across them. We measured the overall throughput (defined as the average of all applications' throughput normalized to their ideal throughput) and the soft memory usage. We compared Midas with three different baselines. The first baseline *overprovisions* soft memory for each application to cache all of possible soft state. This leads to a 67.5 GiB soft memory usage that is impossible to achieve under 20 GiB idle memory; thus, this represents the ideal throughput. The second baseline limits itself to the 20 GiB soft memory budget and *statically partitions* it across four applications in an even manner (*i.e.*, each application gets 5 GiB soft memory). The third baseline is *Cliffhanger* [52]. Similar to Midas, it dynamically coordinates soft memory among applications. However, it adopts a different coordination policy of maximizing the global cache hit rate as opposed to maximizing the overall performance utility. As the original version of Cliffhanger only supports Memcached, we emulated Cliffhanger by implementing its coordination policy atop Midas.

A good result for Midas would show that it quickly reaches an equilibrium by judiciously coordinating soft memory usage among applications and achieves good overall throughput

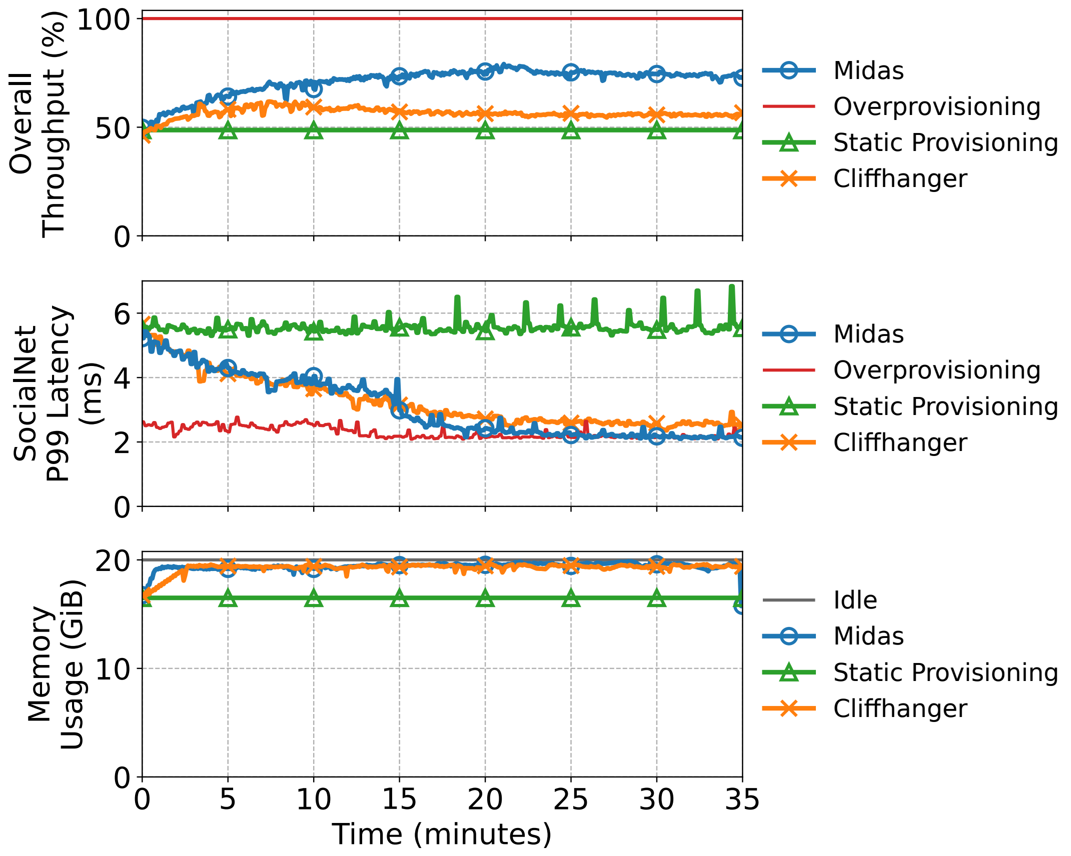


Figure 3.6: When co-running four applications with 20 GiB idle memory, Midas dynamically coordinates their soft memory budgets and reaches an equilibrium in around 20 minutes. Overall, it harvests 19.6 GiB idle memory as soft memory and achieves 75.0% of the ideal throughput (measured by overprovisioning soft memory for all applications regardless of the 20 GiB total budget constraint).

close to the ideal throughput (of the overprovisioning baseline). In contrast, the overall throughput of the static provisioning baseline should be suboptimal, as it equally treats all applications and fails to prioritize the soft memory need of applications that can benefit the most. On the contrary, Cliffhanger does coordinate soft memory among applications, but it optimizes for the overall cache hit rate which does not guarantee optimal overall performance (§3.4.3.2). Therefore, we expect Cliffhanger to achieve overall throughput better than the static baseline but worse than Midas.

Figure 3.6 shows the results. The top figure presents the overall throughput of four systems normalized to the ideal value. The bottom figure presents soft memory usage; we leave out the usage of the overprovisioning baseline as it is much higher (67.5 GiB) than the amount of idle memory (20 GiB). Midas's overall throughput converges in around 20 minutes and achieves 75.0% of the ideal throughput by harvesting 98.0% idle memory. It also reduces SocialNet's 99th percentile latency by 58.4% from 5.5ms to 2.3ms. In contrast, the static provisioning baseline only achieves 48.7% of the ideal throughput and fails to improve SocialNet's tail latency due to the lack of coordination. It also uses 3.1 GiB less soft memory than Midas as some microservices of SocialNet fail to fully use their statically-provisioned soft memory budgets due to small soft memory footprints. Cliffhanger uses a similar amount of soft memory to Midas. Due to its coordination policy, it converges on the overall cache hit rate (not shown due to the space constraint) but oscillates in terms of the overall throughput. Therefore, it only achieves 56.0% throughput on average.

Figure 3.7 presents the per-application soft memory usage of Midas and Cliffhanger. For each application, the gray line represents the soft memory budget it receives, while the colorful line represents the amount of soft memory it uses. Because of the difference in their coordination policies, Midas and Cliffhanger make very different allocations of soft memory between applications except for the storage server. For example, since it is time-consuming to reconstruct HDSearch's objects (as it involves recomputing the feature vectors of images), Midas scales up HDSearch's soft memory to cache more objects. However, since HDSearch has a relatively low request skewness (compared to other applications) and consequently a lower cache utility (in terms of hit rate), Cliffhanger deprioritizes it by scaling down its soft memory, significantly impacting its performance (and therefore the overall performance).

In summary, the experiment demonstrates that Midas can efficiently utilize available memory as soft memory and judiciously coordinate soft memory among applications, achieving high overall performance close to the ideal one that requires 3.4× more memory.

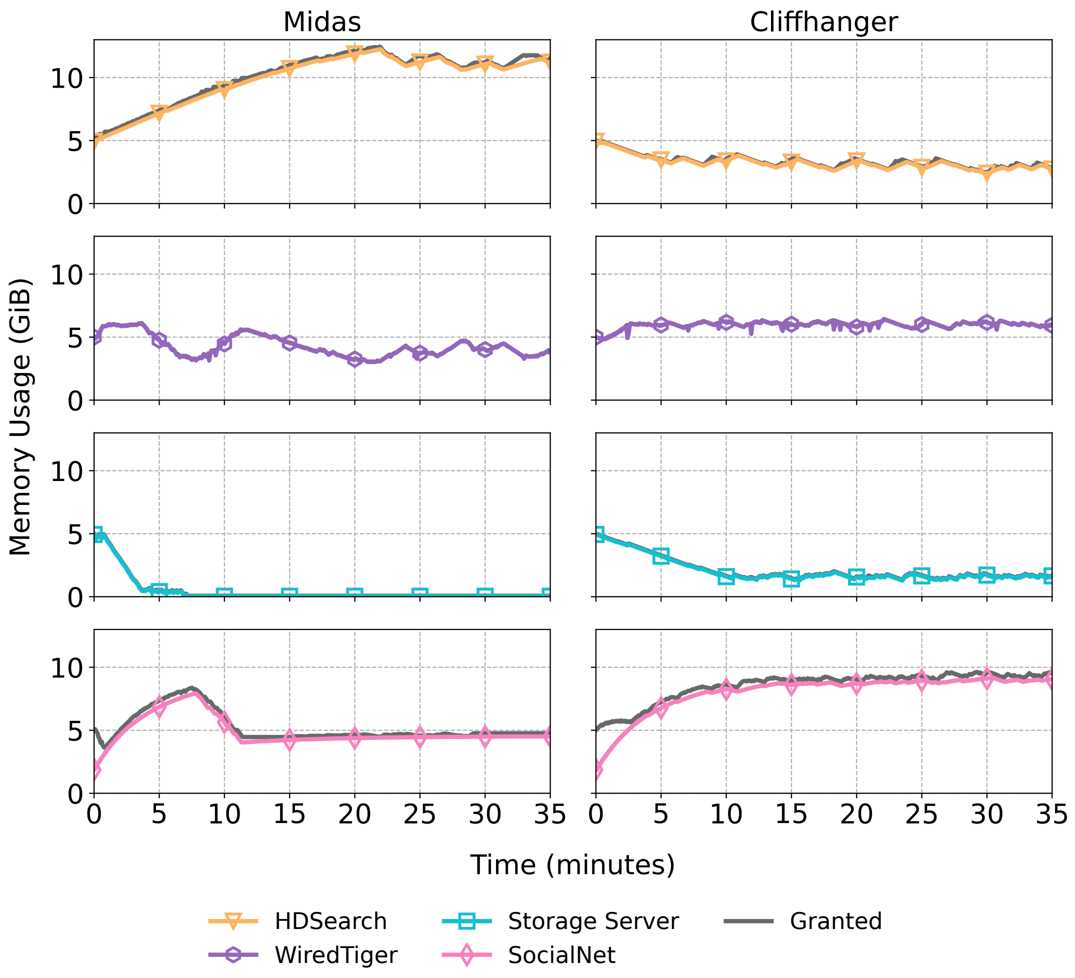
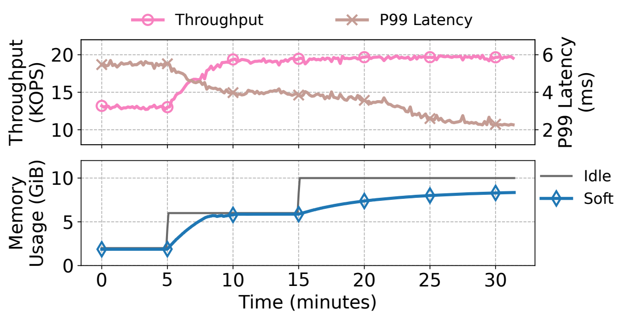


Figure 3.7: Midas and Cliffhanger converge to different allocations of soft memory between applications because of fundamental differences in their coordination policies.

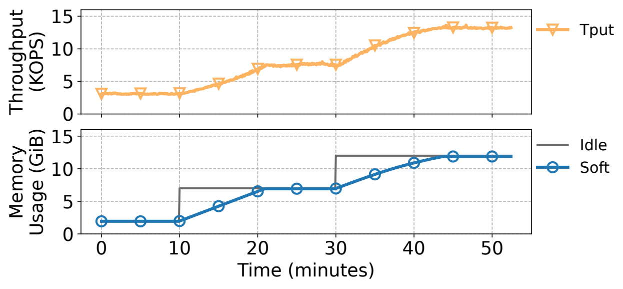
3.7.2 Harvesting Available Idle Memory

In this experiment, we investigated whether Midas can quickly and reactively harvest additional idle memory—whenever it is available—to improve memory utilization and application performance.

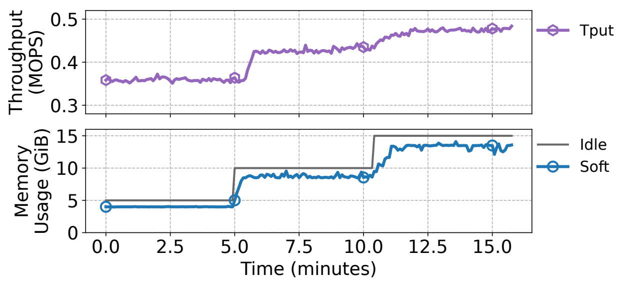
We ran each of the four applications using Midas and dynamically added idle memory to the server. A good result for Midas would show that it quickly detects any new idle memory and reactively grants it to the application as additional soft memory to improve performance.



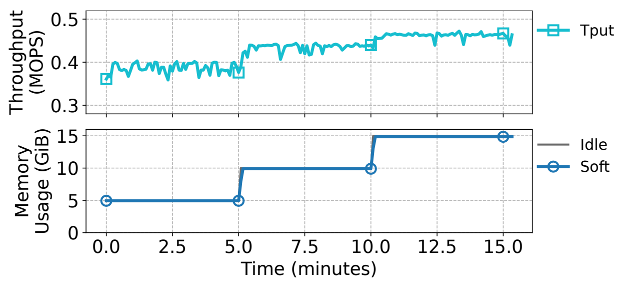
(a) SocialNet.



(b) HDSearch.



(c) WiredTiger.

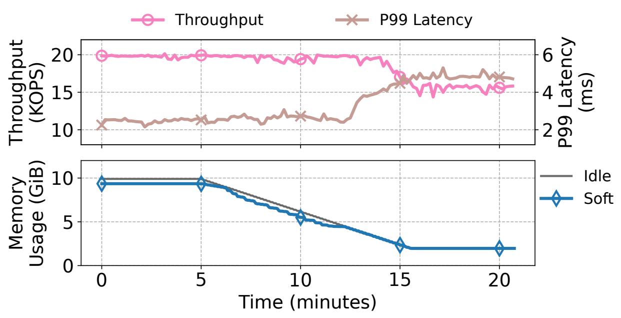


(d) Storage server.

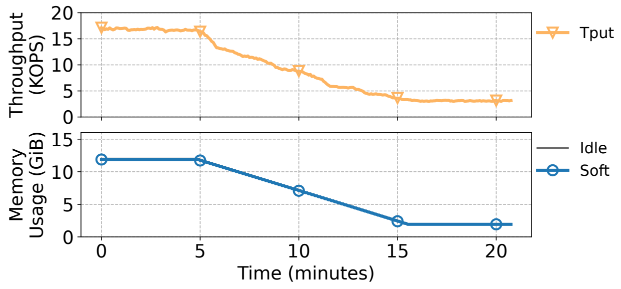
Figure 3.8: With Midas, applications effectively harvest additional idle memory by scaling up their soft memory usage, improving both throughput and tail latency.

Additionally, we expect that the marginal benefit decreases as the application uses more soft memory and caches more hot items.

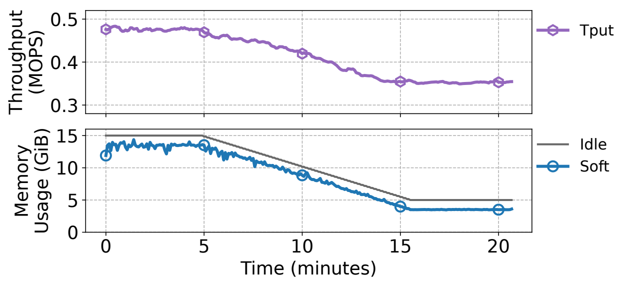
Figure 3.8 presents the results of all four applications. We focus on SocialNet (Figure 3.8a) for detailed analysis, as the other applications demonstrate similar trends, leading to the same conclusion. Initially, the server has 2 GiB idle memory (the dark gray line). With Midas, SocialNet fully utilizes them as soft memory (the blue line) and achieves 13 MOPS throughput (the pink line). At t = 5min, we added 4 GiB more idle memory to the server. Midas immediately detects this change and rapidly ramps up its soft memory usage; it only takes around 3 minutes for SocialNet to reach a new steady state. Benefiting from more soft memory, SocialNet's throughput increases by 46% from 13 MOPS to 19 MOPS, and its 99$^{\text{th}}$ percentile latency decreases by 27% from 5.5ms to 4ms (the light brown line). At t = 15min, we again added 4 GiB more idle memory. This time we observed a reduced



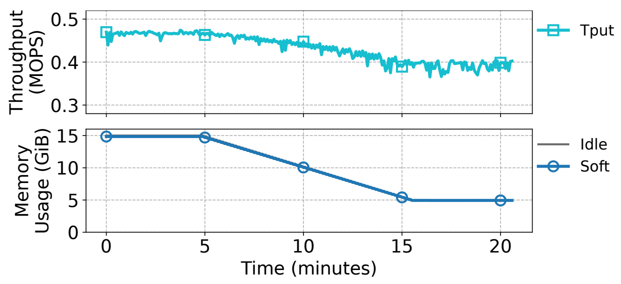
(a) SocialNet.



(b) HDSearch.



(c) WiredTiger.



(d) Storage server.

Figure 3.9: Under moderate memory pressure (t = 5min-15min), Midas is able to reactively scale down each application's soft memory usage to avoid running out of memory with moderate performance impact.

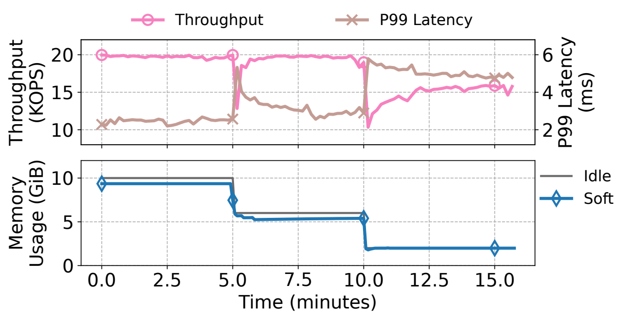
marginal benefit as SocialNet has already cached most hot items; it takes 15 minutes to reach a new equilibrium (*i.e.*, 8.5 GiB soft memory usage) and yields a 43% improvement of 99th percentile latency (from 4ms to 2.3ms).

In summary, these results highlight Midas can quickly detect idle memory and reactively scale up its soft memory usage to improve memory utilization and application performance.

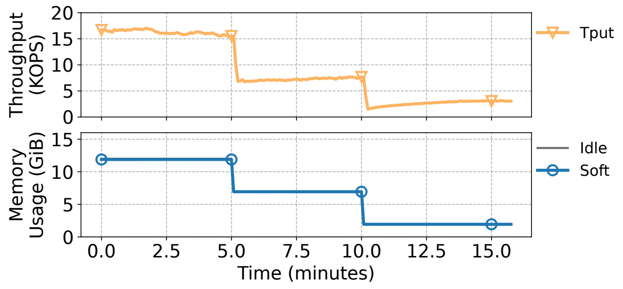
3.7.3 Reacting to Memory Pressure

In this experiment, we investigated whether Midas can quickly react to memory pressure to avoid out-of-memory killing and studied its impact on application performance.

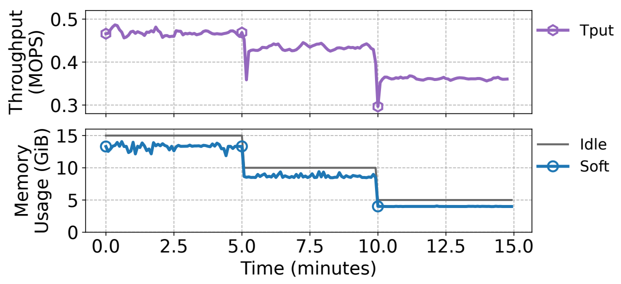
Similar to §3.7.2, we ran each of the four applications using Midas, but dynamically



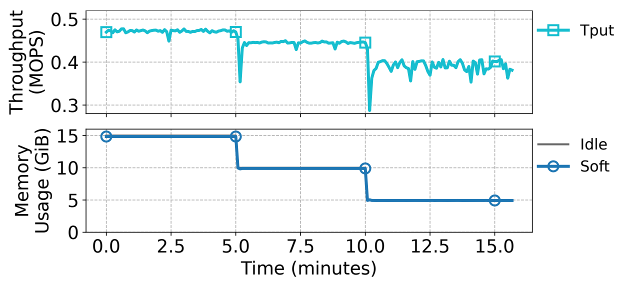
(a) SocialNet.



(b) HDSearch.



(c) WiredTiger.



(d) Storage server.

Figure 3.10: Midas is able to avoid out-of-memory killing even under extreme memory pressure (t = 5min and t = 10min). The victim application experiences brief throughput collapses and tail latency spikes but quickly recovers to normal once the pressure is finished.

decreased the server's idle memory with a colocated memory antagonist. We measured the impact on the victim application's soft memory usage and performance.

Under moderate memory pressure, ideally, Midas's global coordinator should reactively unmap free soft memory segments while Midas's evacuator should be able to replenish them (by evicting cold objects and evacuating hot objects) to match the coordinator's unmapping rate. A good result for Midas would show that the victim application's throughput degrades gradually and mildly as the pressure persists, since Midas prioritizes the eviction of cold and dead objects over hot objects.

Under intense memory pressure, we expect the coordinator to also unmap the used soft memory segments as the evacuator cannot keep up with the high unmapping rate. In this case, the victim application may experience a sudden throughput collapse due to the loss of hot objects. However, a good result for Midas would show that the victim application is still able to operate without experiencing any out-of-memory killing. In addition, immediately after the pressure is finished, the victim application's performance should be able to recover to normal by reconstructing back hotter objects and evicting colder objects.

Figure 3.9 presents the results under moderate memory pressure. Similarly, We focus on SocialNet (Figure 3.9a) for detailed performance analysis, and the other applications demonstrate similar trends. Initially, the server has 10 GiB soft memory. The application uses around 9.6 GiB of it as soft memory and achieves around 20 MOPS throughput and 2.3ms 99^{th} percentile latency. At t = 5min, the memory antagonist starts to allocate 8 GiB more memory at a moderate rate of 0.8 GiB/min, resulting in the decrease of idle memory until t = 15min. As shown by the bottom figure, Midas is able to reactively scale down SocialNet's soft memory usage through reclamation to avoid running out of memory. As shown by the top figure, SocialNet's throughput and 99^{th} percentile latency remain unaffected in the beginning, as Midas prioritizes the reclamation of cold soft memory. After running below 5 GiB idle memory, SocialNet experiences a mild throughput drop and latency increase, as Midas starts to reclaim hotter soft memory.

Figure 3.10a presents the results under intense memory pressure. In this case, the antagonist allocates memory as fast as Linux permits (7 GiB/s), making it an extremely challenging case to handle. Despite the high rate, Midas is still able to avoid out-of-memory killing by rapidly scaling down SocialNet's soft memory usage. In this case, Midas has to unmap the used soft memory segments, inevitably causing brief throughput collapses and latency spikes (at t = 5min and t = 10min). However, once the memory pressure is finished, SocialNet's throughput and latency quickly recovers to the normal level, consistent with the numbers reported in Figure 3.8a and 3.9a.

In summary, these results demonstrate that Midas can always quickly react to memory pressure to avoid out-of-memory killing while maintaining good application performance

[*read/write*] Latency (cycles)	Average *read/write*	Median *read/write*	P90 *read/write*
C++ `unique_ptr`	367 / 199	382 / 176	510 / 332
`SoftUniquePtr`	400 / 393	370 / 368	516 / 500

(a) Small objects (32 B).

[*read/write*] Latency (Mcycles)	Average *read/write*	Median *read/write*	P90 *read/write*
C++ `unique_ptr`	0.97 / 1.39	0.94 / 1.36	0.99 / 1.37
`SoftUniquePtr`	1.77 / 1.15	1.75 / 1.14	1.77 / 1.18

(b) Large objects (4 MiB).

Table 3.2: Midas' soft pointer only adds moderate dereferencing cost compared to C++'s ordinary smart pointer.

whenever it is possible.

3.7.4 Design Drill-Down

Soft Pointer Dereference Cost. We measured the latency of dereferencing a soft pointer and compared it to the latency of dereferencing an ordinary C++ `unique_ptr`, when the pointer and data pointed to are originally in memory (*i.e.*, not in CPU's cache). Table 3.2 shows the results of small objects (32 B) and large objects (4 MiB).

For small objects that fit into CPU's cache line (Table 3.2a), Midas is able to deliver comparable read latency as its extra object copying overhead is negligible. Midas achieves higher write latency (< 200 cycles) as it has to additionally update the metadata in the object header.

For large objects (Table 3.2b), Midas achieves ≈800K cycles (82%) higher read latency

Live Object Ratio		10%	30%	50%	70%	90%
Reclamation	Cooperative	312.5	243.1	173.6	104.2	34.7
Tput. (MiB/s)	Direct			8268.5		

Table 3.3: Midas's cooperative reclamation reclaims memory at the throughput of 35 MiB/s-313 MiB/s, depending on the live object ratio of soft memory. Midas's direct reclamation trades off reclamation quality for faster speed; it achieves a throughput of 8269 MiB/s, exceeding the rate at which the Linux kernel can allocate memory.

since now the additional object copying happens in memory (rather than in CPU's cache). However, Midas achieves lower write latency than `unique_ptr` thanks to its optimized memory copy implementation.

Memory Reclamation Speed. We measured Midas's memory reclamation throughput using a synthetic microbenchmark. Under moderate memory pressure, the coordinator reclaims memory with the cooperation from the runtime (Figure 3.9a); we refer to it as *cooperative reclamation*. Under severe memory pressure, the coordinator directly unmaps soft memory segments (Figure 3.10a); we refer to it as *direct reclamation*.

Table 3.3 presents the throughput of both reclamation methods. The speed of cooperative reclamation depends on the live object ratio of soft memory; the lower the live ratio, the easier to make room by compacting hot objects, thereby yielding faster reclamation speed. It achieves a throughput of 313 MiB/s under 10% live ratio and 35 MiB/s under 90% live ratio. To handle intense memory pressure, direct reclamation trades off reclamation quality for faster reclamation speed; it achieves a significantly higher throughput of 8269 MiB/s, unrelated to the live object ratio. This exceeds the rate at which the Linux kernel can allocate memory (7-8 GiB/s measured in our machine), therefore Midas can always safely harvest server's idle memory without leading to OOM killing.

Performance Impact of Data Reconstruction. To examine the performance impact of using soft memory, we conducted an experiment using a synthetic application; we measured its performance with varying data reconstruction costs under different soft memory ratios (*i.e.*, the ratio of cached soft state). Intuitively, the cheaper the data reconstruction, the lighter the performance impact it incurs.

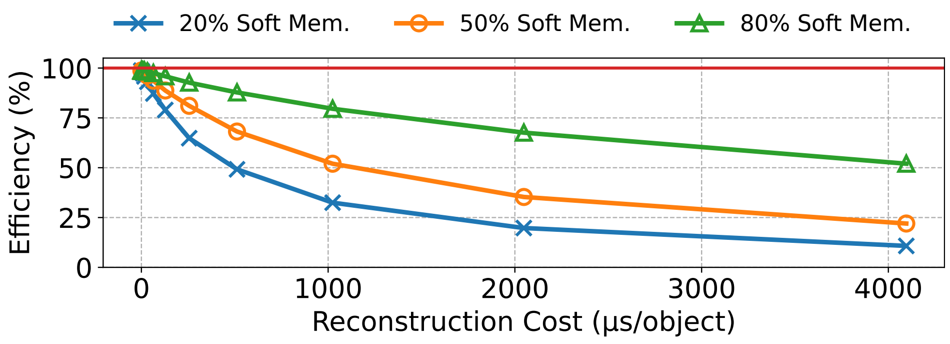


Figure 3.11: Midas's efficiency (y-axis) as a function of data reconstruction cost normalized to the ideal throughput of caching all soft state. Midas's efficiency increases as the reconstruction cost decreases, delivering >80% efficiency for applications with <1024 µs/object reconstruction cost when caching 80% of soft state.

Figure 3.11 shows the result. When the soft memory ratio is 20%, Midas is able to deliver >80% efficiency when the reconstruction cost is <128 µs/object. When the soft memory ratio is higher, Midas can tolerate a higher cost as reconstruction happens less frequently; thus, it is able to provide >80% efficiency for applications with <256 µs/object reconstruction cost under 50% memory ratio and <1024 µs/object under 80% memory. This suggests that Midas can still achieve high performance with moderate data reconstruction costs.

3.8 Related Work

Resource Harvesting and Deflation. Datacenters today suffer from low resource utilization [16, 78, 253]. To make use of vacant resources, major cloud providers now offer spot VMs [15, 87, 164], which run at a low priority and get evicted under resource pressure.

Others propose new VM designs to gracefully adjust VMs' resource usage. Harvest VM is a new type of VM that grows and shrinks according to the amount of unallocated resources at its underlying server, including CPU [16], memory [78], and storage [212]. Similarly, deflatable VM [225] codesigns the hypervisor, VM, and the application to reclaim resources from applications under memory pressure. These approaches focus on VMs only, and take minutes to re-configure a VM to release resources.

Resource Disaggregation and Remote Memory. Resource disaggregation and remote memory systems are another trending approach for improving utilization, thanks to faster datacenter networking [81, 137, 162]. Their key idea is to break the server hardware boundary with a fast network interconnection to exploit stranded resources on a remote server. Various systems have established the viability of disaggregated storage [102, 124], accelerators [184, 254], and memory [12, 89, 224, 260]. While some provide remote memory transparently via OS paging, it is also possible to use a library-based approach that modifies the application to bypass the OS. AIFM [216] proposes remote-able data structures to build remote-memory-aware applications. Semeru [258], Mako [152], and MemLiner [259] co-design the JVM with the kernel to offer transparent remote memory for Java programs. Like Midas, these systems adopt customized pointer formats for their remote-able objects. Unlike Midas, they do not consider the unmap-and-reconstruct semantics and suffer from swapping or out-of-memory killing under intense memory pressure. In Chapters 4 and 5, we will explore remote memory systems in greater detail, providing comprehensive performance analyses and comparisons to highlight these differences.

Cache Services. Improving cache performance is important to datacenter applications, especially in a shared setting [31, 206]. Fairride [206] and RobinHood [31] provide fair and latency-aware cache-sharing policies, and CliffHanger [52] uses a hill climbing method to incrementally optimize cache allocation across applications. Memshare [53] further improves the cache partitioning with a log-structured allocator for higher hit rates. However, existing cache service systems still rely on static memory allocation, and cannot efficiently use idle memory. CacheLib [30] provides a library-based approach for caching, but it again relies on static provisioning and lacks global coordination, hindering its ability to manage memory across multiple applications.

Cooperative Memory Revocation. In parallel with our work, researchers are also exploring the benefits of soft state by managing it at the application level [77]. Midas instead uses kernel coordination and unmap-and-reconstruct semantics, which enables it to reclaim pages even if applications do not cooperate or are slow to respond. This makes it possible to react to severe memory pressure without running out of memory.

3.9 Summary

In this chapter, we presented Midas, a system that efficiently and safely harvests idle memory to store the soft state that is most beneficial to each application, improving both memory utilization and application performance. Midas provides familiar high-level programming abstractions and maximizes overall performance through coordination between an application-integrated runtime and a global coordinator. Our evaluation demonstrates that Midas is able to effectively use soft memory to achieve near-optimal performance and can respond to extreme memory pressure fast enough to avoid running out of memory.

While the main focus of Midas is harvesting local memory within a server, the soft memory abstraction is not restricted by the physical server boundaries. A promising direction for future work (see §7.1) is to extend Midas across multiple servers, which allows applications to benefit even more by harvesting idle memory in the entire cluster. However, to use Midas across servers, developers must first rewrite the single-machine applications into their distributed variants, which requires significant programming effort. In the next chapter, we explore an alternative approach: leveraging the OS kernel's paging and swap mechanisms, but redesigning them for high performance, enabling applications to harness idle memory on remote servers efficiently without code changes.

CHAPTER 4

Hermit: Transparent and Fast Remote Memory Harvesting

In Chapter 3, we explored the use of Midas to exploit soft state semantics for local memory harvesting. While Midas offers superior efficiency and flexibility, it is limited by the physical boundaries of a single machine, preventing applications from scaling beyond a single server. However, the large number of servers within a cluster, together with their potentially imbalanced loads, suggests significant untapped opportunities to aggregate idle resources across the cluster and use them to scale applications.

In this chapter, we revisit the second insight discussed in Section 1.2, that we can achieve much higher aggregated resource availability and capacity if combining idle resources across servers. To mitigate the potentially high programming efforts to port a single-machine application to its distributed alternative, we piggyback on existing OS kernel swap systems, which allow unmodified applications to transparently swap to idle memory on another server to scale out. To overcome the high kernel software overheads, we rethink the OS kernel swap system in the era of fast datacenter network, and redesign it with a novel technique called *adaptive, feedback-directed asynchrony.* These efforts result in Hermit, a new kernel swap system that takes non-urgent but time-consuming operations (*e.g.*, swap-out, `cgroup` charge, I/O deduplication, *etc.*) off the fault-handling path and executes them asynchronously. Additionally, Hermit collects runtime feedback and uses it to direct how asynchrony should be performed—*i.e.*, whether asynchronous operations should be enabled, the level of asynchrony, and how asynchronous operations should be scheduled. These contributions enable Hermit to reduce the tail latency by three orders of magnitude and also substantial throughput improvement for real datacenter applications without changing a single line of code.

4.1 Introduction

Techniques enabling datacenter applications to use remote memory [12, 36, 81, 89, 129, 158, 216, 224, 258] have gained traction due to their potential to break servers' memory capacity wall, thereby significantly improving datacenters' resource utilization. Compared to clean-slate techniques [36, 216] that provide new primitives for developers to efficiently manage remote memory, swap-based techniques [12, 89, 161, 224, 258, 259] that piggyback on existing paging/swap mechanisms in the OS kernel are more practical as they offer transparency, allowing legacy code to run *as is* on a far-memory system.

The main drawback of swap-based remote access is the overhead incurred by the kernel's paging system. For example, when running Memcached using Fastswap [12], the current state-of-the-art swapping system for Linux, a remote access takes an average of *14 μs*, of which only *9 μs* are spent on network (RDMA) operations—the software-induced overhead is *above 50%*! This large fault-handling overhead significantly increases operation latency, precluding the use of remote memory with latency-critical applications.

In addition, long remote-access time can further block subsequent instructions dependent on these accesses, leading to substantial reductions in application throughput. For example, the performance of garbage collection in a managed language runtime is highly sensitive to memory access latency due to its pointer-chasing nature. Reductions in GC performance can lead to delayed object creations, dramatically reducing the application's overall throughput [152, 258, 259].

The underlying reason for such high overhead is a mismatch in the design of today's swap-based paging systems, which originally targeted slow, disk-based storage, and modern datacenter networks (*e.g.*, 100-400 GbE) that can deliver pages much faster. For example, through profiling, we reveal the following performance bottlenecks that persist in Linux (§4.2):

- **Page reclamation blocks the critical path**: To make room to fault in new pages, the OS must reclaim memory by swapping out cold pages. Linux is designed to handle this asynchronously by swapping out pages in a separate thread. However, when Linux fails to keep up with the demand for new pages, the page fault handler must block and wait for reclamation to finish.
- **Duplication checks are too conservative**: Linux is designed to never make duplicate I/O requests for the same page. Although this occasionally prevents wasted bandwidth, it comes at a high cost in terms of synchronization overhead, such as during swap cache lookup and insertion.
- **Opportunities for batching are not exploited**: Batching can be an effective optimization when it does not harm page fault handling latency. For example, when Linux performs page reclamation, it first selects a set of victim pages and then swaps out each page individually. A better strategy would be to process victim pages in batches, reducing the cost of TLB shootdowns, I/O writes, and `cgroup` accounting.

State of the Art. The conventional wisdom is that software overheads can be overcome by bypassing the kernel [202, 216, 278]. This approach typically requires application-level modifications or the use of custom APIs, making it impractical to deploy transparently across all applications. Our aim is to answer the following question instead: *Can we eliminate performance bottlenecks in the kernel directly, allowing the benefits of fast remote memory to be exposed to all applications transparently?*

Recent work, such as Fastswap [12] and InfiniSwap [89], has made some progress in optimizing the kernel's swap subsystem, such as the use of RDMA to deliver remote pages more efficiently. Fastswap, the current state-of-the-art, also modifies the Linux Kernel to offload page reclamation to a dedicated core and executes it asynchronously. This increases swap-out efficiency, and reduces the time that a page fault handler must block waiting for reclamation to finish. However, Fastswap leaves other opportunities for asynchrony on the table. In addition, a single, dedicated core is insufficient to accommodate changes in demand for swap-out throughput under time-varying memory pressure, limiting the conditions where Fastswap can perform well (§4.2).

Insights. This work builds on three insights, all centering around asynchrony. First, asynchrony can be used to reduce the latency of page fault handling. For example, during a page fault, the kernel first looks for the page in the swap cache. If the page is present, it will be mapped at the faulting address and the kernel does not need to issue a fetch. However, this check is protected by a lock, which incurs a non-trivial overhead. Instead, fetching a page via RDMA, even if the page is already in the swap cache, is extremely fast: its only penalty is slightly wasted network bandwidth (*i.e.*, bandwidth is rarely saturated). By always issuing the fetch asynchronously and overlapping it with the check, we can reduce the fault-handling latency.

Second, only page faults handlings are latency critical, so it is safe to aggressively optimize all other operations for throughput via batching. For instance, when TLB shootdowns are batched, it reduces the number of interrupts that have to be sent across cores. As another example, RDMA writes of multiple swapped-out dirty pages can be batched into a single transfer. These opportunities are only possible because such operations are conducted asynchronously; otherwise, batching would delay critical swap-in operations.

Third, to achieve optimal performance, the use of asynchrony (e.g., number of cores) must be adjusted dynamically. For example, it is critical that swap-out throughput is perfectly balanced with swap-in throughput. If swap-out throughput is too low, the page fault handler will block and delay the application. If it is too high, it will leave a substantial portion of local memory underutilized, impacting application performance. This is especially challenging because the swapping rate depends on the workload, its inputs, and even the different phases within its execution.

Hermit. This chapter presents Hermit, a new paging/swap system that exploits these (previously-unknown) opportunities for asynchrony. Hermit employs *feedback-directed asynchrony* as the major principle in the paging system design, simultaneously enabling full code transparency (*i.e.*, any legacy code can run *as is*), low remote access latency, and high application throughput. Hermit employs different types of asynchrony to tackle the three bottlenecks (*i.e.*, blocked swap-ins, conservative checks, lack of batching), as elaborated below:

First, page reclamation is moved into a set of reclaim threads, which eagerly evict (least-recently used) pages and aggressively batch expensive operations involved in each swap-out (§4.3.2). In particular, Hermit batches page unmapping, TLB shootdown, RDMA writes, polling, and `cgroup` uncharging in swap-out threads, reducing the amounts of computation involved in swap-outs and improving their throughput (§4.3.4).

Second, Hermit opportunistically bypasses the swap-in duplication check and issues I/O read requests eagerly, delaying such checks to the synchronous PTE update stage. Since only one thread can successfully update the PTE, all other competing threads will eventually release their duplicate pages, guaranteeing safety (§4.3.3).

Third, inspired by optimistic locking [134], Hermit makes page I/O fully asynchronous during swap-in to further reduce latency. We split the swap-in procedure into two components: one that can still successfully run and is reversible even if there are concurrent updates, and a second that may either abort or create irreversible side effects in the presence of concurrent updates. Hermit moves the first component out of the critical section to overlap it with the page I/O (details are in Figure 4.4). Hermit checks the validity before the critical section finishes (*i.e.*, whether concurrent updates have occurred) and if they have, reverts the speculatively executed operations.

Finally, we create a feedback control system for each type of asynchronous operation, using

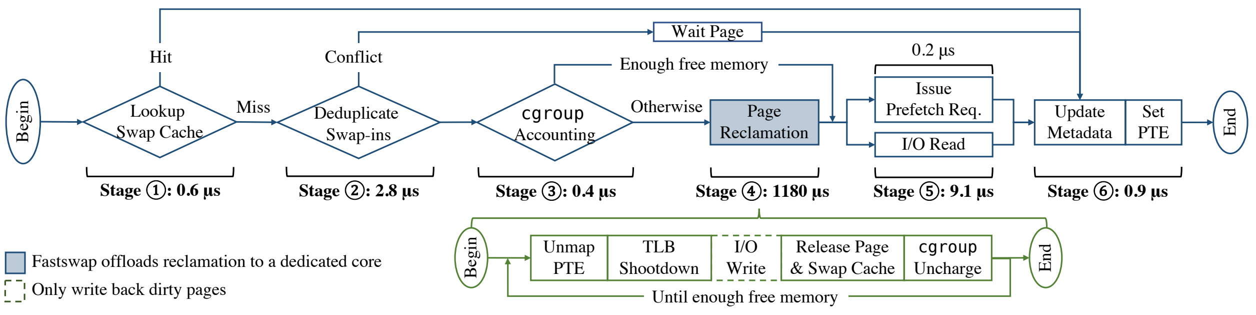


Figure 4.1: The life cycle of a remote memory page fault in Linux swap.

execution profiles to adjust whether and how asynchrony should be applied. In particular, we use (1) *page turnaround* (*i.e.*, time between a page's swap-in and previous swap-out), (2) *page-in/-reclamation throughput*, and (3) *conflict rates* (*i.e.*, how often concurrent updates occur), as metrics to adjust our asynchrony in dealing with reclamation timing, reclamation intensity, eager swap-in, conservative checks, respectively. Hermit profiles and collects these signals throughout the execution to dynamically adapt to the application's changing behaviors.

Results. Hermit was implemented in Linux 5.14. We evaluated Hermit with a set of real-world applications including both latency-critical (Memcached, SocialNet, and Gdnsd) and batch-processing applications (Apache Spark, XGBoost, and Apache Cassandra). Our evaluation on Memcached demonstrates that Hermit outperforms Fastswap [12] by **99.7%** in latency, reducing the 99^{th} percentile latency **from 36 ms to 91 µs**. For batch processing applications, Hermit improves throughput by up to **1.87×** with a geometric mean of **1.24×**. Hermit also scales much better with the number of cores than Fastswap. These results demonstrate that low tail latency and high throughput can be achieved at the same time without bypassing kernel, making Hermit a practical solution for enabling remote memory. Hermit is available at `https://github.com/uclasystem/hermit`.

4.2 Understanding Existing Swap Systems

Section 2.2.1 has provided a high-level overview of the kernel swap system. In this section, we delve deeper by conducting a detailed performance study and breaking down the costs associated with each stage when swapping in a page in the context of remote memory. Following this analysis, we examine Memcached [163], a widely-used cloud cache service, to identify and analyze the root causes of kernel swap inefficiencies in real-world scenarios.

4.2.1 The Life Cycle of Remote Memory Access

The legacy design of Linux swap imposes high overheads on accessing remote memory. To better understand the root cause of its inefficiencies, we conducted a performance study by running Memcached on Fastswap [12] (the state-of-the-art swap system). Figure 4.1 shows the stages of a remote memory access and breaks down their costs. We discuss each stage in more detail as follows:

① **Lookup swap cache.** The swap cache serves as a centralized component that prevents race conditions. It tracks the information of swapped-in pages and ongoing swap-out requests. First, the faulting page may have been fetched by another process or the OS prefetcher. By looking up the swap cache, Linux detects this and jumps to stage ⑥. Second, it is possible that the faulting page is being swapped out by another process. In this case, naïvely fetching the remote page will see the stale copy. With the swap cache, Linux detects the race and cancels the ongoing swap-out. Looking up the swap cache takes an average of 0.6 µs.

② **Deduplicate swap-ins.** At the same time, there can be multiple threads swapping in the same page. Linux guarantees that only one thread can succeed by synchronizing with lock primitives. The remaining threads will be busy waiting until the page gets fetched. This design saves I/O bandwidth but impacts latency and hurts scalability. This stage takes an average of 2.8 µs.

③ `cgroup` **accounting.** Before fetching the page, Linux must ensure that the current process has sufficient free memory by performing `cgroup` accounting. For the lucky process with enough memory, it jumps to stage ⑤ directly. The accounting stage takes an average of 0.4 µs. Otherwise, Linux must go through stage ④ to reclaim pages to make room, as elaborated below.

④ **Direct page reclamation.** Linux iteratively reclaims pages until the size of the available local memory is above the low-water mark. Linux swaps out a single page for each iteration. Swap-out is expensive as it involves operations such as TLB shootdown, PTE unmapping, *etc.* This stage exists only when the local memory runs low, but it is also the longest one that takes an average of 1180 µs. To reduce direct reclamation, Fastswap performs this stage asynchronously with a dedicated core.

⑤ **Fetch and prefetch page.** Linux issues an I/O request to fetch the faulted page. Meanwhile, it may issue multiple prefetching requests. This stage takes an average of 9.1 µs.

⑥ **Update metadata.** Finally, Linux updates kernel metadata, including page table entries (PTEs), swap entries, and page reverse mapping (`rmap`). This stage takes 0.9 µs.

4.2.2 Root Causes of Inefficiencies

To understand the bottleneck imposed by Fastswap's single, dedicated reclamation core, we ran several experiments with Memcached. Figure 4.2 shows Memcached's 99$^{\text{th}}$ percentile latency with respect to its offered load when running with 70% local memory. The baseline for comparison is Memcached running locally (100% local memory without swapping), which is the rightmost curve and achieves >4.4 Mops load throughput with good tail latency. Memcached on Fastswap (the blue curve), however, can only offer ≈1 Mops load before the dedicated core gets saturated and its latency increases dramatically. The reason is that Fastswap's single

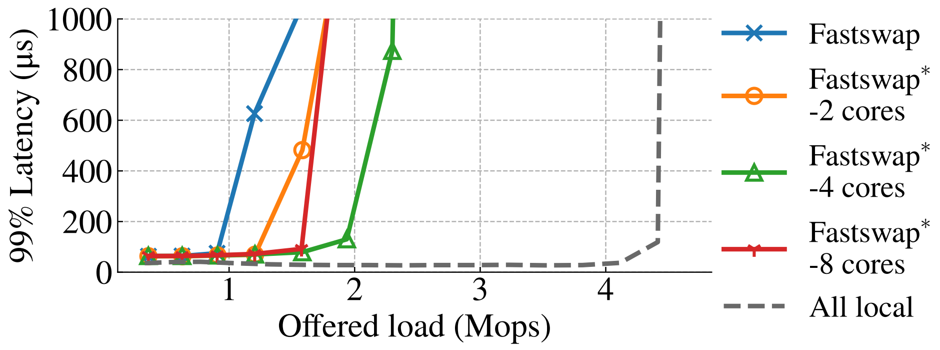


Figure 4.2: 99th percentile latency with respect to offered load of Memcached on Fastswap under 70% local memory.

dedicated core cannot keep up with the increasing demand for page reclamation. We then modified Fastswap's original implementation to offload page reclamation onto multiple cores, denoted as `Fastswap`* in the figure, as a naïve strawman approach.

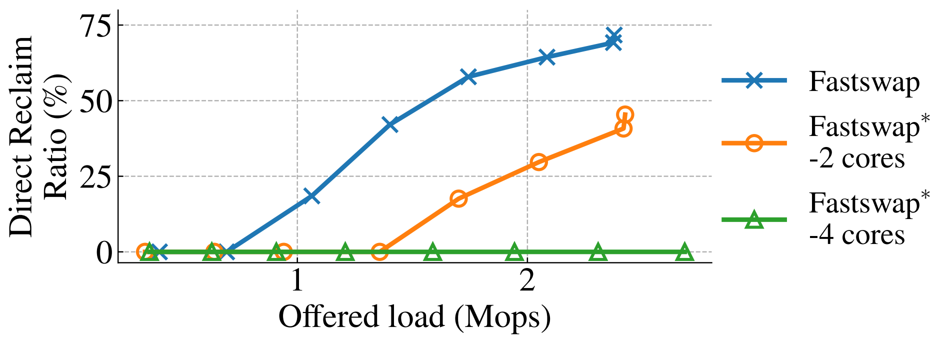


Figure 4.3: Direct page reclamation ratio of Memcached on Fastswap under 70% local memory.

Using more dedicated cores can indeed help reduce the direct reclamation ratio, as shown in Figure 4.3. With 4 dedicated cores, `Fastswap`* is able to eliminate direct page reclamation, thus providing the highest throughput among all Fastswap variants. However, Fastswap uses static core provisioning, which is insufficient in practice due to the phased behaviors and shifts in load that occur within datacenter applications. First, the number of required dedicated cores depends on the application's working set, the available local memory, and the swap-in intensity, making it impossible for a statically determined number to work universally for different applications or even different phases of the same application. Second, over-provisioning dedicated cores does not always lead to greater end-to-end performance; in many cases, using more cores only shifts the bottleneck from page reclamation to the application itself, as more dedicated cores for reclamation imply fewer available cores for application threads. As shown in Figure 4.2, increasing the number of dedicated cores in Fastswap from 1 to 4 (`Fastswap*-4 cores`) improves performance, but further allocating cores degrades performance (`Fastswap*-8 cores`). Furthermore, although `Fastswap*-4 cores` eliminates direct page reclamation (*i.e.*, reducing latency), it still loses ~45% performance (*i.e.*, reducing throughput). The performance loss is due to three major kinds of inefficiencies induced by Linux swap, as elaborated below.

Swap-out blocks swap-in. As explained earlier, Memcached experiences high memory access latency when running short of local memory, as it has to reclaim pages. Page reclamation is expensive as it requires finding victim pages and unmapping them, followed by a number of expensive operations for consistency such as TLB shootdown. This significantly impacts its tail latency, leading to violations of the service-level agreement (SLA).

Fastswap tackles this issue by allocating a dedicated core to reclaim pages asynchronously in the background. However, as discussed earlier, it is nearly impossible to statically identify the optimal number of cores due to load variability.

Unoptimized for fast I/O. Linux swap was designed for slow secondary storage like hard-disk drives whose performance is two to three orders of magnitude lower than today's remote memory in both bandwidth and latency. Since disk bandwidth is often the bottleneck, Linux applies aggressive optimizations in its page fault handling path to reduce I/O traffic (stage ②). While they were effective in the era of slow disks, these optimizations become irrelevant in the context of remote memory whose bandwidth is close to the bandwidth of main memory. Even worse, the outdated optimization generates an adversarial performance impact; it prolongs remote memory access latency, hurting scalability (*e.g.*, due to synchronization).

For latency-critical applications like Memcached, prolonged remote memory accesses can significantly increase the time for serving incoming requests, imposing super-linear effects on tail latency. Modeled by queueing theory [66], for instance, 10% longer service time can potentially double the 99$^{\text{th}}$ percentile latency, leading to vast SLA violations.

Additionally, since the disk latency (ms-scale) is significantly higher than the CPU time in page fault handling (µs-scale), Linux adopts a serial-execution model for simplicity. As shown in Figure 4.1, the I/O read stage is executed separately from other stages; after issuing the I/O read request, Linux either busy waits for the I/O response or re-schedules the faulting thread (which hurts latency of fast I/O requests), relinquishing the opportunity of overlapping the waiting period with other stages.

Unoptimized for CPU overhead. Linux swap is a mechanism aimed at avoiding OOM killing. Inherently, treating swapping as a rare event, it was designed to optimize for responsiveness, *not* for CPU efficiency. For example, during page reclamation (stage ④), Linux swaps out only one page at a time, under the assumption that by releasing the space more timely it can unblock the OOM process sooner. Unfortunately, this amplifies the CPU usage as it must invoke expensive operations such as TLB shootdown for *every reclaimed page*. While overhead is acceptable when swapping is rare, it grows significantly in the scenario of remote memory (which is swapping-intensive). In the case of Memcached, 12.6% of the total CPU time is spent on reclaiming pages, not on application tasks. To make matters worse, Linux swap heavily relies on locks to synchronize page reclamation and scales poorly. Hence, the overhead will further increase with the number of concurrent swapping operations.

Key takeaway. Linux swap imposes high overheads to remote memory access primarily due to the above three issues. Fastswap, the state-of-the-art swap system, partially tackles the first issue, but neglects the last two. For the first issue, Fastswap uses statically provisioned cores to run swap-out tasks; as shown in Figure 4.2, static core provisioning cannot adapt to dynamic load changes, leading to either insufficient or wasted CPU resources.

4.3 Hermit Design

4.3.1 Design Overview

To overcome the aforementioned inefficiencies, we developed Hermit, a new swap system based on the principle of *feedback-directed asynchrony.* Our key insight is that asynchrony should be used aggressively (to overlap nonurgent and urgent operations to reduce latency), but this must be done in a controlled manner—whenever asynchrony cannot bring benefits, we should switch back to the conventional synchronous design. Figure 4.4 illustrates Hermit's design.

First, Hermit optimizes tail latency of accessing remote memory by moving page reclamation from the critical path into the background (§4.3.2). Instead of following the design of Fastswap, which statically reserves a certain number of dedicated cores, Hermit relies on a *reclaim scheduler* to dynamically schedule reclaim threads. The scheduler leverages feedback from `cgroup` counters to determine the right timing and the appropriate number of cores for reclamation.

Second, the swap-in path of Hermit was designed with fast remote memory in mind (§4.3.3)—for remote memory, it is reasonable to trade off network usage for end-to-end performance as modern datacenter network offers abundant bandwidth (100-400 Gbps). In the common case, Hermit detects idle network bandwidth and opportunistically bypasses swap-in duplication checks (stage ② in §4.2) to improve scalability and reduce latency. This bypassing has a consequence: in the (rare) case that multiple threads are fetching the same page at the same time, they will all transfer the same page over the network. Note that this will *not* lead to correctness issues because only one copy will be mapped by the PTE in the last stage, and any other requests will abort and release their page. However, it may potentially waste some network bandwidth when duplicate pages are requested. Therefore, instead of bypassing blindly, we use the conflict rate (in the last stage) as a control signal to determine whether it is beneficial to enable bypassing. To further optimize the critical-path latency, Hermit also overlaps the I/O read stage with other swap-in operations (*e.g.*, `cgroup` accounting, metadata updating, *etc.*).

Finally, we structured Hermit to operate in a swap-intensive environment to match the reality of using remote memory (§4.3.4). Hermit carefully optimizes the CPU usage of page reclamation so that more CPU resources are available for applications. Enabled by Hermit's reclaim scheduler, which reduces the "urgency" of reclamation tasks, Hermit opportunistically handles reclamation requests *in batches* to amortize the overhead. In addition, Hermit bypasses the expensive reverse mapping operation when swapping out a private page (which is common). As a result, Hermit not only reduces the remote access latency but also significantly improves the application's throughput.

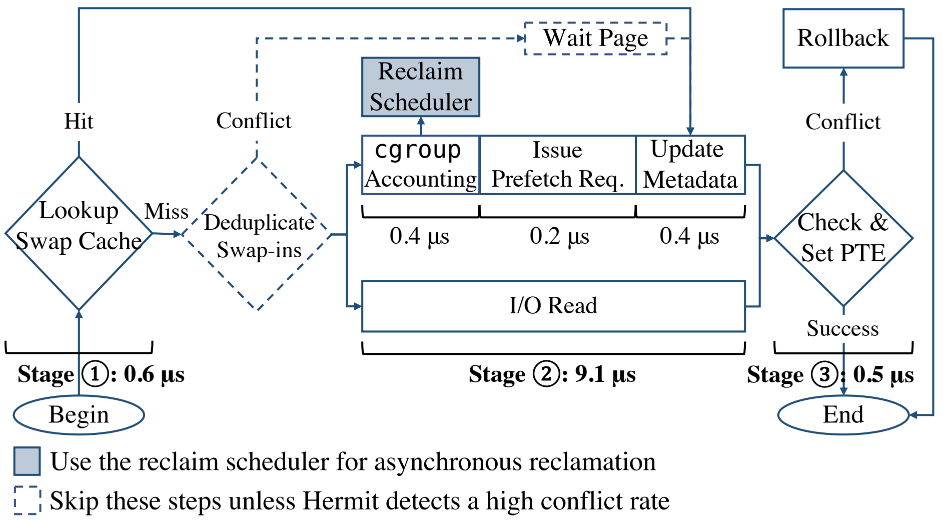


Figure 4.4: The life cycle of a remote memory page fault in Hermit.

4.3.2 Reclaim Scheduling

In Linux swap, the direct page reclamation in the swap-in path significantly impacts the tail latency of accessing the remote memory. To reduce tail latency, Hermit moves reclamation off the critical path into background threads; the reclaim scheduler monitors the free memory size and *proactively* starts reclamation before memory exhaustion. The scheduler uses the application's swap throughput as a feedback signal to auto-tune the number of reclaim threads.

Designing such a scheduler is challenging because it must determine both the right *timing* and the appropriate *amount of CPU resources* for reclamation. (1) As for the timing, if the scheduler starts reclamation too early, a substantial portion of local memory would be underutilized, impacting application performance; on the flip side, if the scheduler starts reclamation too late, the application would exhaust the local memory and suffer from direct reclamation. (2) As for CPU resources, under-provisioning cores for reclamation (*i.e.*, the case of Fastswap) make it unable to keep up with the local memory consumption rate, leading to memory exhaustion, while over-provisioning cores is also undesired as it contends with the application and reduces its performance.

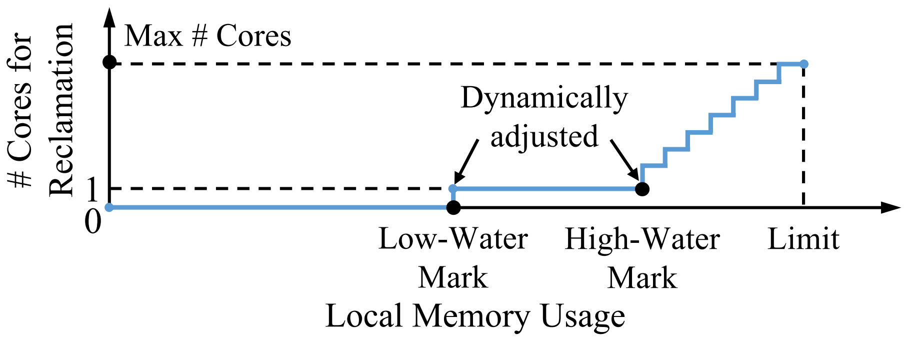


Figure 4.5: Adaptive reclaim scheduler.

Figure 4.4 shows the design of the reclaim scheduler, which leverages counters from `cgroup` to schedule reclamation. Since the timing for reclamation is critical to performance, our reclaim scheduler has to be very reactive to free memory size changes (in µs-level). Instead of using a dedicated core to poll the memory usage which waste CPU cycles, Hermit adopts a *decentralized* reclaim scheduler; it inlines the scheduler code into the `cgroup` charging, an indispensable step for swap-ins. This design enables us to discover any sudden change in the free memory size with only a few CPU cycles.

Hermit's scheduling policy follows the conventional wisdom of random early detection

[72] to gradually increase its asynchronous reclaim throughput. Specifically, Hermit starts asynchronous reclamation when application's memory budget is running low, but Hermit will only enable a small number of reclaim threads first and gradually increase the number of reclaim threads after observing constantly increasing memory usage. The intention of the design is to handle a burst of swap-ins within the memory limit with as few reclaim threads as possible, and thus minimizing asynchronous reclamation's interference to the application.

On the other hand, when the application is about to run out of memory, Hermit must unleash the full power of asynchronous reclaim threads to match the reclaim throughput to swap-in throughput to avoid direct reclamation, offering the application maximum swap performance. Figure 4.5 depicts Hermit's adaptive scheduling policy, which determines the number of cores for page reclamation given the application's current local memory usage. The curve can be divided into three phases, marked by the low-water mark and the high-water mark to differentiate the urgency of asynchronous reclamation.

When the application does not swap intensively and its local memory usage is below the low-water mark, the number of reclamation cores is zero, indicating that the asynchronous page reclamation is disabled now to let application threads have all CPU cores. When the application's local memory usage is between the low-water mark and the high-water mark, it indicates that the application is under memory pressure, and the scheduler will assign one core for asynchronous reclamation to relieve the memory pressure with minimal compute to minimize its interference to application's threads.

Finally, when the application hits the high-water mark, it indicates that the application is about to run out of memory. Page reclamation is an urgent task now to prevent the application from triggering direct page reclamation. As such, the reclaim scheduler must assign more cores for reclamation to match the reclaim throughput with application's swap-in throughput. As Figure 4.5 shows, during this phase, the number of cores assigned for reclamation is proportional to the local memory usage, reaching the maximum value when the local memory usage equals the memory limit.

Hermit leverages the kernel's runtime statistics to auto-tune the low and high memory watermarks, as elaborated below.

High memory watermark. Hermit dynamically adjusts the high memory watermark based on the application's current *swap intensity*. We define swap intensity as the overall swap-in throughput divided by the per-core page reclamation throughput, representing the number of cores needed for reclamation to match the swap-in speed. Intuitively, when the swap intensity increases, we should lower the high-water mark to start ramping up reclamation earlier; and when the swap intensity decreases, we should raise the high-water mark accordingly. Hermit sets the high-water mark as $MEM\_LIMIT - \alpha \cdot SWAP\_INTENSITY$, where $\alpha = 128$ works well in practice.

Low memory watermark. Initially, Hermit sets the low-water mark to be the same as the high-water mark. Then it gradually probes its optimal value based on the average page turnaround time (APT), defined as the average duration for swapped-out pages to remain untouched. When APT does not increase, Hermit attempts to lower the low-water mark, as now it can potentially start reclamation earlier without impacting the application performance. However, when APT increases, Hermit immediately raises back the low-water mark to revert the negative impact on the application performance.

4.3.3 Adapt Swap-in to Fast Remote Memory

As shown in Figure 4.4, Hermit re-architects the swap-in path for the fast remote memory with two main innovations.

Eager swap-in. Hermit opportunistically bypasses the swap-in duplication check to minimize latency. As such, it is now possible that multiple threads issue swap-in requests for the same page. To ensure that only one of them will succeed, Hermit synchronizes them in the final stage (updating PTE) using a fine-grained lock. All other failed threads will release their swapped-in pages—CPU cycles consumed by them are wasted and considered as penalty. Hermit collects the conflict rate and the penalty as feedback to reassess whether it is still beneficial to enable eager swap-in and disable it if it impacts performance.

Asynchronous I/O. Hermit further shortens the critical path of swap-ins by overlapping the I/O read with other operations, for example, `cgroup` charging. If later the `cgroup` check shows no memory, Hermit discards the I/O read response and updates the failure counter. Hermit falls back into synchronous I/O when the failure ratio is high. This happens very rarely in practice thanks to Hermit's asynchronous reclamation (§4.3.2).

4.3.4 CPU-Efficient Page Reclamation

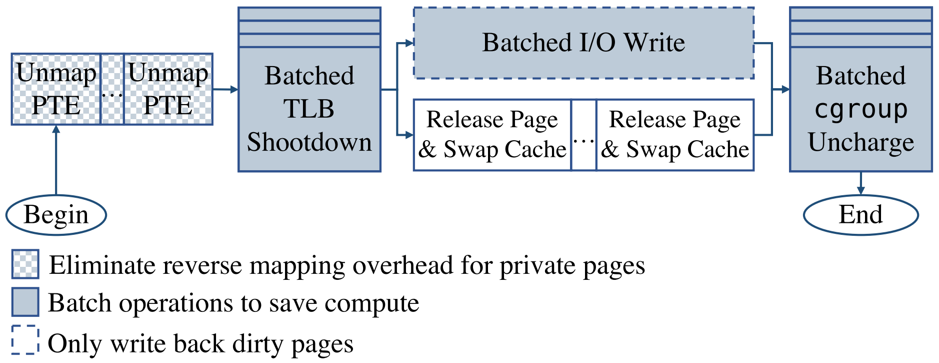

Unmap PTE
…
Unmap PTE
Batched TLB Shootdown
Batched I/O Write
Release Page & Swap Cache
…
Release Page & Swap Cache
Batched cgroup Uncharge
Begin
End
Eliminate reverse mapping overhead for private pages
Batch operations to save compute
Only write back dirty pages

Figure 4.6: Hermit's asynchronous page reclamation path.

As shown in Figure 4.6, Hermit carefully optimizes the CPU overheads of page reclamation to minimize its performance impact to applications.

Batched reclamation. As illustrated in §4.2, Linux's page reclamation is mainly designed for slow disk devices where swapping occurs infrequently—it trades off CPU efficiency for responsiveness by only swapping out one page at a time. However, Hermit overcomes the responsiveness loss with its asynchronous reclamation design, which relaxes the responsiveness requirement of page reclamation, thereby creating opportunities for batching. As depicted in Figure 4.6, Hermit batches expensive operations, including TLB shootdowns, I/O writes, and `cgroup` accountings—to amortize their overheads in the asynchronous page reclamation path.

Reverse mapping elimination. To avoid race conditions during reclamation, Linux has to ensure that the page is immutable before writing it back to remote memory. Linux achieves this goal by using `rmap` (reverse page mapping) to identify and unmap all the virtual pages mapped to the reclaimed physical page. `rmap` walk is expensive as it involves several memory accesses and lock synchronizations. A key observation in Hermit is that most reclaimed pages are private pages (*i.e.*, only referenced by one virtual page). For private pages, Hermit eliminates the expensive `rmap` walk by inlining the virtual page address into the physical page metadata in Linux. This approach trades a tiny portion of local memory (0.2% in the worst case) for better performance.

4.4 Implementation

We implemented Hermit atop Linux 5.14. We added or modified 9704 lines of kernel code, mainly re-implementing Linux's swap-in and swap-out code paths.

We built our RDMA-based swap backend atop Fastswap's implementation. The original Fastswap uses Linux's `frontswap` interface which only supports blocking I/O. We extended it with an asynchronous I/O interface to enable asynchronous batched I/O writes during page reclamation.

For the swap-in path, we stored the feedback signals `swap_stats`, used by Hermit to decide whether to bypass the swap-in deduplication, in Linux's process context `mm_struct`. `swap_stats` contains two atomic counters representing the numbers of successful and aborted swap-ins respectively. The page fault handler reads and updates `swap_stats` when swapping in the page.

For the swap-out path, we implemented per-`cgroup` reclaim threads as Linux kernel threads. We stored the feedback signals `swap_ctrl`, used by Hermit to decide the swap-out timing, in Linux's memory cgroup `mem_cgroup`. `swap_ctrl` contains two counters representing the total number of charged pages and reclaimed pages. Hermit updates `swap_ctrl` during `cgroup` charging and page reclamation. The reclaim scheduler reads `swap_ctrl` periodically (per 128 charges in our implementation) to calculate the swap intensity for updating the high-water mark. We use Linux's existing mechanism of tracking the page re-fault distance to calculate the average page turnaround (APT) for updating the low-water mark. Hermit batches 32 pages per NUMA node for its asynchronous page reclamation to keep low amortized overheads while ensuring most reclamations can finish timely (within 1 ms). To batch reclamation while ensuring consistency, we carefully ordered the operations (see Figure 4.6). Hermit first selects and unmaps a batch of pages, and then issues a single TLB flush before writing all dirty pages to remote memory. After which, Hermit rechecks each page to ensure it remains untouched and free it. Otherwise, the page must have been faulted on and re-mapped into the process' page table, so Hermit skips freeing this page and returns it back to the application. To bypass the `rmap` walk, we stored the virtual address of private pages using a global array. We did not directly embed the virtual address into Linux's per-page metadata to avoid breaking its cache alignment.

4.5 Evaluation

Our evaluation seeks to answer the following questions:

1. Can Hermit maintain low tail latency (§4.5.2) and high throughput (§4.5.3) while delivering remote memory?
2. How does Hermit's performance compare to standard Linux and Fastswap [12]? (§4.5.2-§4.5.3)
3. What contributes to Hermit's better performance? (§4.5.4)

Setup. We ran experiments in a cluster with one CPU server and one memory server, connected by a 100 GbE network. Each server equips a 24-core AMD 7402P CPU and 128 GB memory. Both Hermit and Fastswap ran on Ubuntu 20.04 with Linux 5.14. For latency-critical applications, we generated load from another server, which connects to the CPU server via a 25 GbE network. We followed common practices to tune these servers for low latency [194], including disabling CPU frequency scaling, machine-check exceptions, and transparent hugepages. We also disabled OS security mitigations as recent CPUs have fixed these vulnerabilities. We enabled hyperthreading as it improves the performance of remote memory systems.

Methodology. We compared Hermit with the ideal system that only uses local memory and the state-of-the-art kernel-based remote memory system, Fastswap [12]. To enable a fair comparison, we also ported Fastswap to Linux 5.14, the same kernel version that Hermit uses.

4.5.1 Real-world Applications

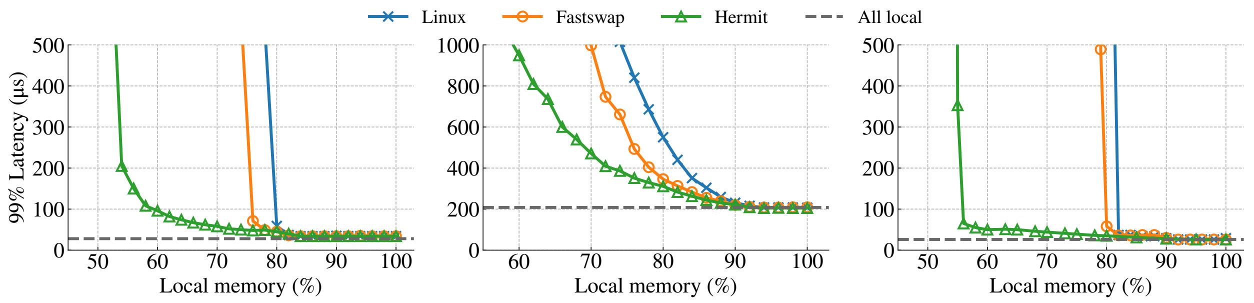
We used six real-world datacenter applications for evaluation, as shown in Table 4.1.

Category	Application	Dataset	Size
Latency-Critical	Memcached [163]	Facebook's `USR` [22] like	32M KVs
	SocialNet [80]	Socfb-Penn94 [214]	41.5K nodes, 1.4M edges
	Gdnsd [82]	Custom	75M sites
Batch-Processing	Spark [274]	Wikipedia EN [126]	188M points
	XGBoost [48]	HIGGS [24]	21M instances
	Cassandra [18]	YCSB [56]	20M records

Table 4.1: Applications used in the evaluation.

Latency-critical applications. Memcached [163] is a popular in-memory key-value store. It only performs a hash table lookup for each request, leading to a small per-request memory footprint. It has low compute intensity and poor spatial locality. We followed Facebook's `USR` distribution to generate load with 99.8% GET and 0.2% PUT [22]. SocialNet (a part of the DeathStarBench [80]) is a twitter-like interactive web application built with microservices. It has a fan-out pattern in which each client request is served by multiple microservice instances. This leads to a larger per-request memory footprint than Memcached. It has medium compute intensity and poor spatial locality. We rewrote DeathStarBench's python-based load generator using C++ to increase its throughput. Gdnsd is an authoritative-only DNS server. It performs a tree lookup for each DNS query. It has a small per-request memory footprint and low compute intensity. Different from previous applications, Gdnsd has good spatial locality. We generated queries with random domain names for evaluation. For all three applications, we generated requests with keys followed Zipf distribution using the skewness parameter $s = 0.99$, to be consistent with the standard YCSB benchmark suite [56].

Batch applications. Apache Spark [274] is a big data analytics engine. We used the logistic regression model from its official example suite for evaluation, in which Spark trains the model iteratively by scanning the dataset to update the model parameters. It has high compute intensity and a large memory footprint. XGBoost is a gradient boosting library for machine learning. We ran binary classification for evaluation. It initializes a group of decision trees and trains them iteratively by splitting the tree leaves with input data. It has dynamic parallelism and a medium memory footprint. Apache Cassandra [18] is a large-scale NoSQL database. It uses a storage structure similar to a log-structured merge tree, which has medium compute intensity and good spatial locality. Different from other batch applications, it also periodically persists in-memory data to disk. We used YCSB [56] as its workload for evaluation. Both Spark and Cassandra are Java-based and run atop OpenJDK-11. Java's garbage collection makes them more memory intensive. XGBoost is a native C++ application.



(a) Memcached (2 Mops) (b) SocialNet (0.75 Mops) (c) Gdnsd (4 Mops)

Figure 4.7: Hermit significantly outperforms Fastswap and Linux in terms of 99% latency under the same fixed load and varying local memory ratio. Hermit enables applications to operate in a more challenging regime of less local memory while still maintaining < 500 µs 99% latency.

4.5.2 Tail Latency of Latency-Critical Applications

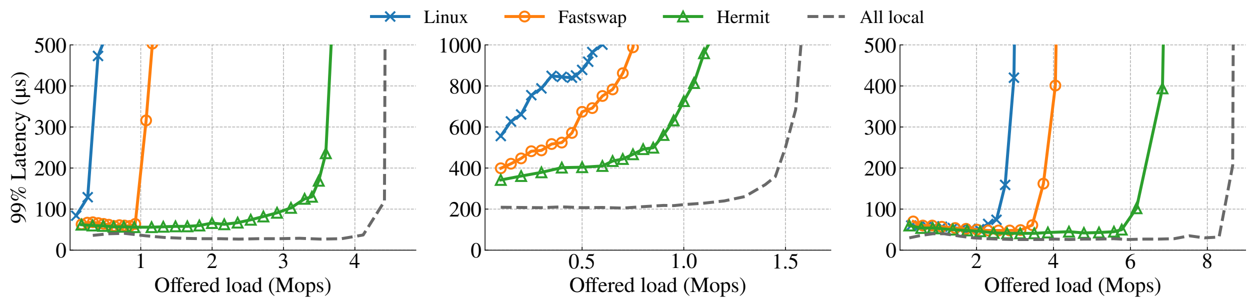
To better quantify the tail latency overhead introduced by Hermit, we use low-latency applications enabled by Shenango (a recent datacenter library OS) [194], for evaluation. With Shenango's low-latency threading runtime and network stack, these applications achieve sub-millisecond tail latency, making it an extremely challenging case for swap systems. We also rerun the same applications with their vanilla (Linux-based) versions. Following previous studies [64, 118, 278], we primarily focus on applications' 99^{th} percentile latency in our evaluation. A more detailed evaluation, including the results of other percentiles such as median and 99.9^{th}, can be found in the original paper [207].

We first ran applications with a fixed load (50% of load capacity measured with only using local memory) and varying local memory ratios. We measured the application performance on Linux, Fastswap, Hermit, and the ideal setup that only uses local memory (see Figure 4.7). The original Linux does not have an RDMA-based swap backend. To enable a fair comparison, we extended it to use Fastswap's RDMA backend. On Figure 4.7, the X-axis shows the ratio of the local memory provisioned; the Y-axis shows the 99^{th} percentile latency achieved by

Linux, Fastswap, Hermit, and the ideal setup.

Intuitively, both Fastswap and Hermit achieve ideal performance when only using local memory. When we decrease the local memory ratio, latency increases as remote accesses become more frequent. However, Hermit's latency increases slower than Fastswap, revealing it is more tolerant to remote accesses. This is because Hermit's overhead of accessing remote memory is lower, thanks to its shorter swap-in path and its reclaim scheduler that eliminates direct reclamation (§4.5.4.1). As Hermit adaptively changes the number of reclaim threads to match the reclamation rate with the swap-in rate, it can result in competition for CPU resources if the local memory ratio is small enough. Eventually, both systems encounter a "hockey-stick" when they cannot handle the excessive remote memory accesses. Compared to Fastswap, Hermit enables applications to operate in a more challenging regime of less local memory while still maintaining < 500 µs 99th percentile latency.

Specifically, the low compute intensity of Memcached and Gdnsd aligns with Hermit's optimizations well; they only require a few CPU cores for serving load, leaving the rest of the cores for reclamation. Moreover, thanks to their small per-request memory footprints, they only require a small number of reclaim threads. For Memcached, Hermit has to rely on more than four reclaim threads to keep up with frequent swap-ins when Memcached runs under $< 60\%$ local memory ratio. The CPU contention gets more severe when local memory gets smaller, and the system reaches 70% CPU utilization under 58% local memory ratio. Afterward, Hermit's reclaim threads can heavily interfere and block Memcached's threads, thus ramping up the tail latency. Similarly, Gdnsd on Hermit used ~72% CPU cycles when running under 56% local memory ratio, and the system can no longer maintain low 99th percentile latency afterward. Fastswap's single dedicated core fails to keep up with the increasing page reclamation demand when local memory ratio is lower than 76% and 82% for Memcached and Gdnsd, respectively, which ramps up their 99th percentile latency. To conclude, Hermit pushes the operating regime in terms of local memory ratio from 75% (*i.e.*, Fastswap) to 55% for Memcached, and from 80% to 55% for Gdnsd. Gdnsd has a slightly



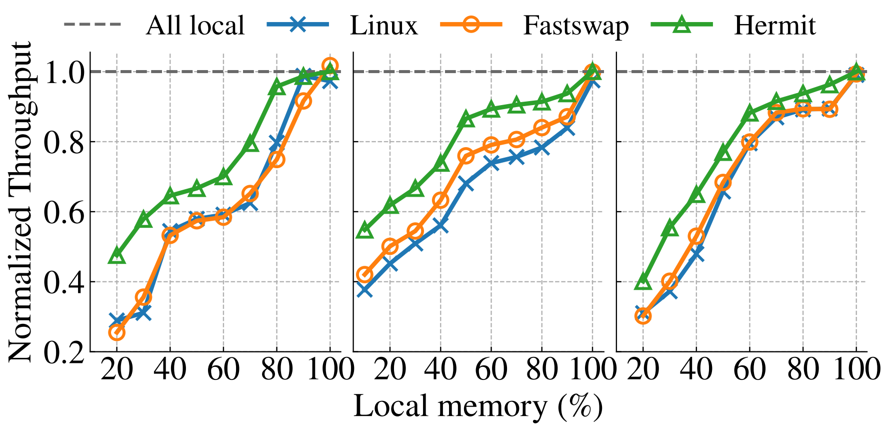
(a) Memcached (b) SocialNet (c) Gdnsd

Figure 4.8: Hermit achieves significantly lower 99% latency than Fastswap and Linux under the same fixed local memory ratio and varying load. For Memcached and Gdnsd, Hermit achieves 99% latency close to the ideal local-only case. SocialNet is more challenging due to its higher per-request memory footprint, but Hermit still achieves 74% load capacity of the ideal case.

better result due to its better spatial locality. SocialNet is a more challenging application that has a higher compute intensity and a larger per-request memory footprint. It requires more reclaim threads which compete with application threads more heavily under low local memory ratios. The system used 70% of its CPU resources under 65% local memory ratio, and saturated all CPU cores under 60% local memory ratio. Hermit pushes its regime from 75% local memory ratio to 65%. In summary, Hermit enables applications to store an average of 20% more working set in remote memory without breaking the tail latency target, thereby harnessing stranded memory resources more efficiently.

Next, we fixed the local memory ratio to 70% and measured the tail latency of applications with varying load (see Figure 4.8). Under low load, both Fastswap and Hermit encounter higher latency than the local-only case due to additional remote memory accesses. Hermit delivers lower latency than Fastswap due to the cheaper remote accesses it offers. For Memcached and Gdnsd whose per-request memory footprint is smaller, Hermit reduces 99$^{\text{th}}$ percentile latency by 3–9 µs, whereas for SocialNet, Hermit reduces latency by 43–86 µs.

Under high load, the latency gap becomes wider because of the CPU contention between



(a) Spark (68.4s) (b) XGBoost (42.2s) (c) Cassandra (72.6s)

Figure 4.9: We measured the throughput of batch applications achieved by different swap systems normalized to the ideal local-only setup. Hermit outperforms other baselines. The number in the parenthesis shows the ideal execution time.

application and asynchronous reclaim threads. In this case, application threads access remote memory intensively, therefore triggering memory reclamation frequently. The asynchronous reclaim threads impact application performance by contending CPU resources. Hermit experiences lower performance degradation because of its asynchronous and more CPU-efficient design of memory reclamation (§4.5.4.2). By eliminating blocking induced by direct reclamation and shifting more CPU resources from reclamation to application, Hermit handles higher load than Fastswap under the same local memory ratio while still maintaining < 500 µs 99th percentile latency. Hermit improves the load capacity by 3.2× (from 1.1 Mops to 3.5 Mops) for Memcached, and 1.7× (from 4.0 Mops to 6.8 Mops) for Gdnsd. Notably, compared to the ideal local-only case, Hermit enables these applications to enjoy the benefit of remote memory with only an average of 20% decrease in their load capacity. It is more challenging to handle SocialNet well due to its larger per-request memory footprint and higher compute intensity. As a result, the number of reclaim threads needed increases quickly with the load, deteriorating the contention with application threads. Even though, Hermit still improves SocialNet's capacity by 1.5× (from 0.75 Mops to 1.15 Mops).

4.5.3 Throughput of Batch Applications

In this section, we evaluate the throughput of batch applications under varying local memory ratios (see Figure 4.9). Hermit outperforms both Fastswap and Linux. It only requires 45%–70% local memory to achieve at least 80% of the ideal throughput for all applications. In contrast, Fastswap (*i.e.* the better baseline) has to use an average of 20% more local memory to achieve the same throughput. Even under the extremely challenging case of 20% local memory, Hermit is still able to preserve 40%–60% of applications' ideal throughput. This leads to 1.23×–1.87× improvement over Fastswap.

When Spark runs atop Fastswap, its throughput drops significantly when running with $<$ 40% local memory. Our profiling reveals that swapping becomes extremely frequent in this case, triggering the scalability bottleneck in kernel's page reclamation path. Hermit does not suffer from the same issue due to two reasons. First, Hermit significantly reduces the direct reclamation ratio by performing reclamation asynchronously and timely. Therefore, it confines reclamation into a small number of reclaim threads rather than all the application threads (in direct reclamation). Second, Hermit's CPU-efficient reclamation design reduces the number of threads needed, further alleviating the scalability issue.

4.5.4 Design Drill-Down

We now evaluate specific aspects of Hermit's design to understand their individual contributions to overall performance.

4.5.4.1 Remote Memory Access Latency

Hermit reduces remote memory access latency by shortening the critical path of swap-ins. Figure 4.10 breaks down the improvements brought by specific optimizations, including bypassing deduplication and using asynchronous I/O. The results are measured using Memcached. Without Hermit's optimizations, the original Linux spends 2.8 µs on swap-in deduplication.

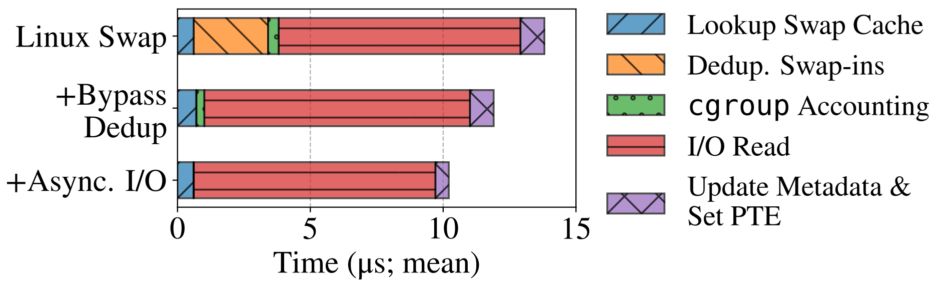


Figure 4.10: Hermit reduces the remote memory access latency in Memcached from 13.8 μs to 10.2 μs with two optimizations, *i.e.*, bypassing deduplication and using asynchronous I/O.

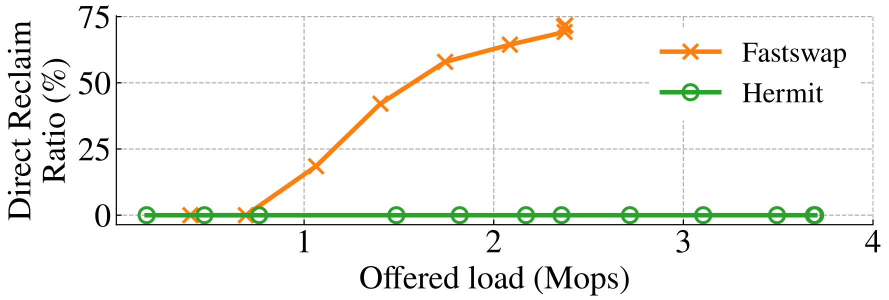


Figure 4.11: Hermit entirely eliminates direct reclamation for Memcached, thanks to its asynchronous reclamation design. Fastswap fails to serve > 2.4 Mops load due to CPU congestion.

Hermit eliminates this overhead entirely by opportunistically bypassing the deduplication, see Figure 4.11. After enabling asynchronous I/O, Hermit further overlaps I/O read with other swap-in operations (*e.g.*, `cgroup` accounting and metadata updating), reducing the swap-in latency by another 0.9 μs. With both optimizations turned on, Hermit reduces the page fault handling latency by 35%, from 13.8 μs to 10.2 μs. The RDMA backend spends 9 μs on performing a 4KiB-page I/O. This indicates that Hermit reduces the overhead of the swap system by a factor of four, from 4.8 μs to only 1.2 μs.

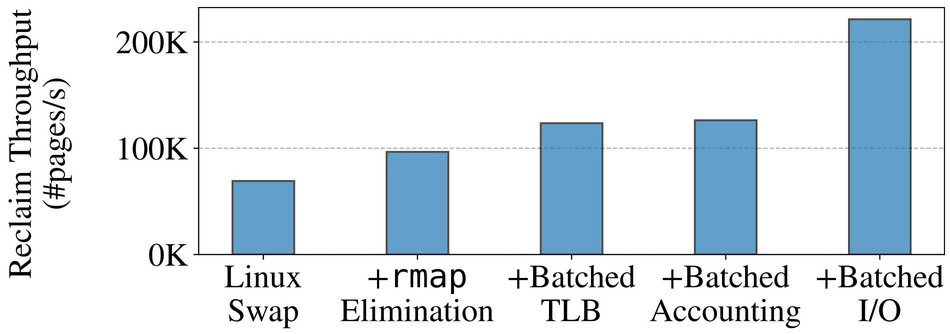


Figure 4.12: Eliminating reverse mappings and enabling more batching makes reclamation 2.9× more efficient.

4.5.4.2 Page Reclamation Efficiency

To demonstrate Hermit's improvements on page reclamation efficiency, we ran Memcached and measured the per-thread reclamation throughput, see Figure 4.12. As shown by the leftmost bar, the original Linux achieves 77K pages/s reclamation throughput. Hermit's `rmap` elimination optimization effectively improves the throughput by 37%, as most of pages are private in Memcached. Batching TLB shootdowns and `cgroup` accountings amoritizes their overheads and brings an additional 27% and 3% improvement, respectively. Finally, Hermit batches I/O writes for dirty pages and overlaps them with the page release phase. This significantly reduces the time wasted on polling for the write completion, generating a 75% further improvement. Our further profiling reveals that Hermit reduces the per-page overhead of `rmap` by 59% from 1.70 µs to 0.69 µs, TLB shootdown by 92% from 2.45 µs to 0.20 µs, and I/O writes by 88% from 6.47 µs to 0.76 µs. To summarize, Hermit improves the single-thread page reclamation throughput from 77K pages/s to 221K pages/s, making reclamation 2.9× more efficient.

4.5.4.3 Effectiveness of Feedback-directed Asynchrony

To demonstrate the importance of Hermit's feedback-directed asynchrony, we modified Hermit's reclaim scheduler to use Fastswap's static scheduling policy. The new version

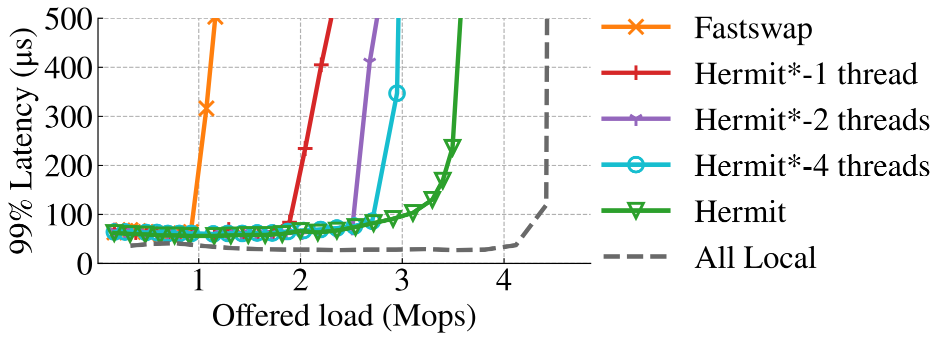


Figure 4.13: Hermit's feedback-directed asynchrony is indispensable for achieving superior performance. Hermit considerably outperforms all `Hermit`*s—the modified versions that adopt Fastswap's static scheduling policy for reclamation.

`Hermit`* uses a fixed number of reclaim threads and starts reclamation only when the free local memory size falls below 8 MiB. Figure 4.13 shows the results of Memcached.

Hermit consistently outperforms all variants of `Hermit`*, regardless of the number of reclaim threads statically configured. Our further profiling reveals that the memory pressure during Memcached's execution varies over time. In most cases, it only requires ≤2 reclaim threads to mitigate the pressure. However, upon sudden bursts of requests, it needs up to 4 threads to fully keep up with the demand. Hermit's reclaim scheduler dynamically adjusts the number of reclaim threads to adapt to the changes in demand, thereby achieving superior performance to its static counterparts.

4.5.4.4 Breaking Down End-to-End Speedup

We evaluated the individual contribution of each of the three optimizations (§4.5.4.1-§4.5.4.3) to the overall application performance.

For latency-critical applications, we used Memcached as the representative. We reran Memcached with the same configuration as Figure 4.8 (a) with optimizations enabled incrementally. Figure 4.14 reports the results. Linux even fails to handle low load of 0.5 Mops under 70% local memory, as it frequently triggers direct reclamation which can easily prolong Memcached's 99th percentile latency by hundreds of microseconds. Fastswap outperforms

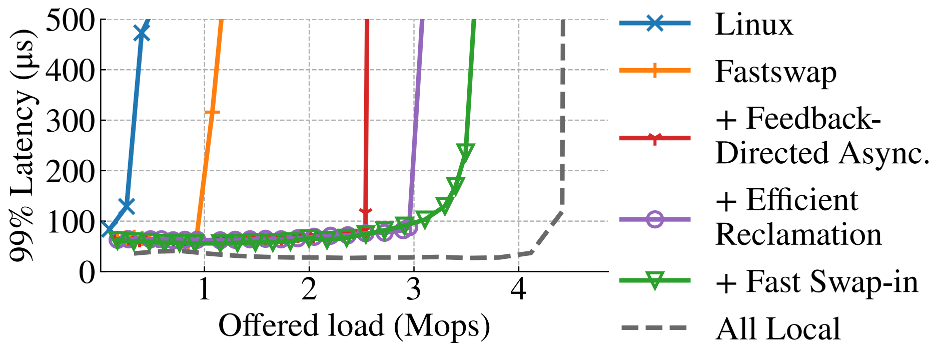


Figure 4.14: All three of Hermit's optimizations work in tandem to improve Memcached's latency and throughput. Results are measured with 70% local memory.

Linux by offloading reclamation to a dedicated core. However, the application quickly saturates the core's reclamation capacity once the load reaches 1.1 Mops, and starts to trigger direct reclamation again (see Figure 4.11). This prevents Fastswap from maintaining low 99th percentile latency afterward.

With the reclaim scheduler (§4.3.2), Hermit can handle a much higher load, 2.5 Mops, before the latency starts to spike. This is because Hermit's reclaim scheduler proactively and timely starts asynchronous reclamation, eliminating the blocking caused by direct reclamation. Optimizations in the reclamation path (§4.3.4) reduce the amount of CPU resources required. This alleviates the contention between reclaim threads and application threads, adding 0.4 Mops to the load capacity. Finally, optimizations in the swap-in path (§4.3.3) make remote memory accesses faster and reduce the per-request processing time, thereby enabling Memcached to achieve higher load with the same amount of compute. Putting them all together, Hermit helps Memcached reach 3.5 Mops using 70% local memory while maintaining 99th percentile latency under 250 µs.

For batch applications, we used Spark as the representative and re-ran it under 20% local memory with the same configuration as Figure 4.9(a). Figure 4.15 breaks down the performance improvements. Our reclaim scheduler again improves the application throughput by a large margin (31%) due to the following reasons. First, batch applications usually follow

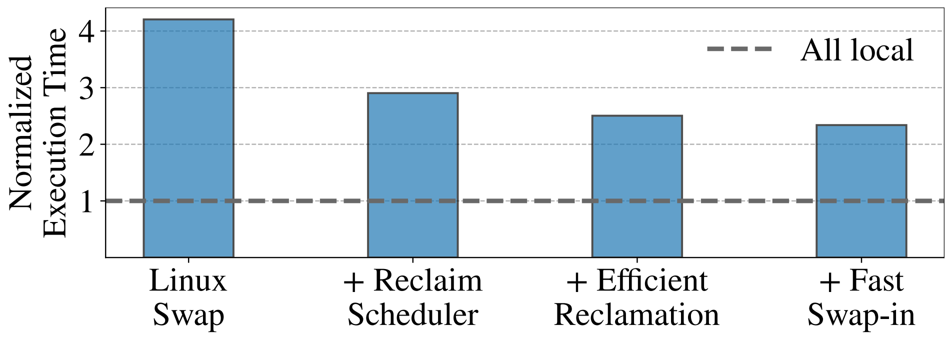


Figure 4.15: All three of Hermit's optimizations collectively improve Spark's throughput. The Y-axis shows the execution time normalized to the ideal local-only time (68.4s). Results are measured under 20% local memory.

the epochal hypothesis [182], whose compute and memory behaviors vary during an epoch but repeat across epochs. Asynchronous reclamation unleashes the hidden parallelism by speculatively reclaiming pages, making it possible for reclaim threads to efficiently harness idle compute resources in each epoch. Second, Linux swap frequently triggers massive direct reclamations instantaneously, causing severe lock contentions between page faults handlings (swap-in) and reclamation. Hermit avoids the burst of reclamation and greatly alleviates the contention by reclaiming asynchronously and proactively. Further, optimizations on the page reclamation path and the swap-in path collectively improve the swap efficiency: they yield an additional 10% and 4% throughput improvement, respectively.

4.5.4.5 Resource Consumption of Swap Operations

Network Bandwidth. Hermit performs swap operations eagerly to improve performance. It opportunistically bypasses swap-in deduplication to reduce swap-in latency (§4.3.3) and proactively schedules asynchronous reclaim threads to avoid direct reclamation (§4.3.2). These optimizations offer performance benefits potentially at the cost of additional network usage. For example, Hermit might swap in the same page several times in the presence of concurrent page faults. To confirm that Hermit does not incur excessive network traffic, we measure the

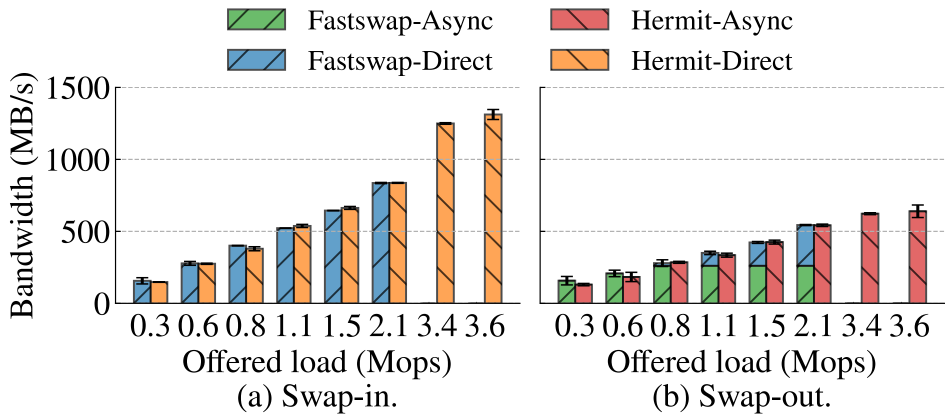


Figure 4.16: Hermit's optimizations do not incur additional network usage during swap-ins/-outs compared to Fastswap.

network bandwidth used for swap-ins and swap-outs, and compare it with Fastswap's usage.

Figure 4.16 shows the results when running Memcached. The X-axis shows the offered load while the Y-axis shows the average network bandwidth. The error bar quantifies the bandwidth fluctuation during the application's execution. With higher offered load, both Fastswap and Hermit use more network bandwidth as Memcached swaps memory more frequently. The bandwidth usage in swap-outs is lower than in swap-in as clean pages do not need to be written back during reclamation.

For swap-in, Hermit incurs similar network bandwidth usage compared to Fastswap. This is consistent with our further investigation which reveals that the conflict rate (*i.e.* the ratio of concurrent page faults that swap in the same page) is less than 0.07%. Therefore, Hermit's swap-in optimization barely introduces any extra network overhead in practice.

For swap-out, we break down the total bandwidth consumption into the usage of asynchronous swap-out and direct swap-out. Hermit is able to constantly perform asynchronous reclamation without using additional network bandwidth compared to Fastswap. This makes sense as Hermit's optimizations to reclamation timing and efficiency do not inflate the number of reclaimed pages.

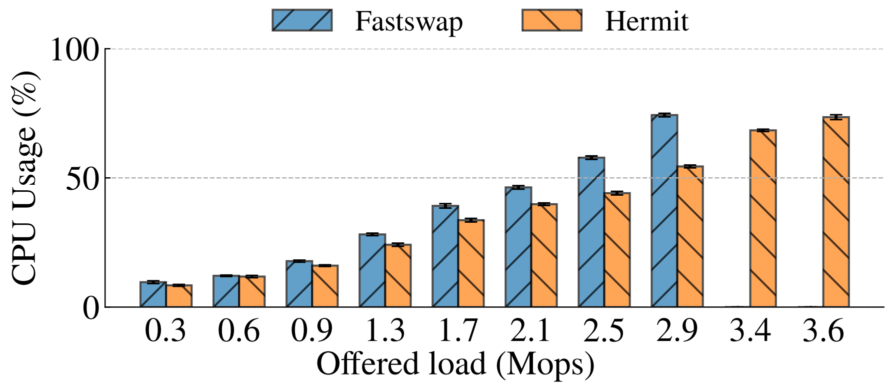


Figure 4.17: Hermit saves ~30% CPU cycles under varying load compared with Fastswap, which is the key enabler to achieve low 99$^{\text{th}}$ percentile latency under high load.

CPU Cycles. We also profiled the CPU usage of applications running on Fastswap and Hermit, revealing that Hermit can serve much higher load with the same amount of CPU resources. Figure 4.17 depicts the total CPU usage of Memcached and Hermit's reclaim threads under 70% local memory ratio and varying load. When increasing load, both Fastswap and Hermit use more CPU cycles as Memcached swaps more frequently. We observed that Memcached fails to use $>$ 70% CPU cycles due to its internal lock contention on hot slabs under skewed workloads. Even though Hermit can spawn more reclaim threads than Fastswap (when needed), it uses 20%–30% fewer CPU resources overall, thanks to its feedback-directed asynchrony and more effective use of batching. Therefore, Hermit is able to offer 32% higher load capacity for Memcached compared to Fastswap.

4.6 Related Work

In addition to the related work covered in §2.2, we categorize the subsequent related work by specific topics.

Kernel-based Remote Memory. To provide transparency to existing applications, the kernel-based approach leverages OS paging to access and manage remote memory. Most kernel-based systems build upon Linux, including Hermit. Infiniswap [89] is an early work that integrates Linux's swap subsystem with an RDMA-based block device backend. Later, Fastswap [12] leverages the lightweight `frontswap` interface to reduce overhead and offloads page reclamation to a dedicated core. Leap [158] improves Linux's prefetcher to achieve a higher local memory hit rate. The ongoing advances of Linux's virtual memory subsystem in the kernel community also benefit Linux-based remote memory. These include, but are not limited to multi-generational LRU [74], speculative page faults [134], maple-tree-based VMAs [100], and DAMON-based proactive page reclamation [196]. Finally, other kernel-based work like LegoOS [224] builds new OS, hoping for better performance with its clean-slate approach.

Library-based Remote Memory. The library-based approach bypasses OS to reduce the kernel overhead and overcome the granularity restriction imposed by paging. It trades application transparency for performance; application developers often have to significantly rewrite their code using the new remote memory APIs. FaRM [68] and KVDirect [135] expose remote memory with an external key-value store interface which mismatches with the construction of existing applications. Distributed shared memory (e.g., [139, 179]), on the other hand, provides an object-oriented interface that is more user-friendly. AIFM [216] proposes remote-able data structures to capture rich application semantics, but requires programmers to re-write the code with its APIs. Semeru [258], Mako [152], and MemLiner [259] are JVM-based remote memory runtimes, offering transparency for Java applications by co-designing the JVM and the kernel.

Hardware-accelerated Remote Memory. Another type of work proposes novel hardware designs, thereby unlocking new opportunities for optimizing remote memory. Hermit focuses on the software layer and benefits from the advance of the underlying hardware. PBerry [35] and Kona [36] overcome the granularity restriction of paging and enable cache-line-level remote memory access. Clio [94], StRoM [235], and RMC [13] reduce the expensive network traffic by offloading tasks into the customized hardware of the memory server. Finally, the emerging CXL bus [137] is promising to lower the bar of accessing remote memory by delivering performance close to local DRAM.

4.7 Summary

In this chapter, we present Hermit, a re-architected swap system that leverages the novel technique of adaptive, feedback-directed asynchrony. Our evaluation shows that Hermit significantly outperforms Fastswap (the state-of-the-art swap system) in real datacenter applications; it reduces the 99th percentile tail latency by 99.7% and improves the throughput by 1.24×. Hermit defies the conventional wisdom about kernel-based remote memory, demonstrating that it is possible to achieve both full transparency and high performance simultaneously.

The success of Hermit also unveils a deeper challenge in today's multi-tenant cloud environments, where servers often colocate multiple applications. However, because the kernel swap system is shared globally by all applications running on a server, it can introduce resource contention and performance interference when multiple applications share remote memory. In the next chapter, we quantify the interference effect and address this challenge by introducing a comprehensive swap isolation and fair share mechanism.

CHAPTER 5

Canvas: Isolated and Adaptive Remote Memory Harvesting

In Chapter 4, we illustrated how Hermit enables an application to scale out and harvest idle resources on a remote server. However, in the real production environment, it is common for multiple applications to co-locate on the same server [106, 282]. Consequently, these applications can share remote memory, including the swap system code path, the network, the local swap cache, and other swap resources. Unfortunately, today's OS kernel swap system is not well prepared for such resource-sharing scenarios, as it focuses on the efficiency of a single application setting only, overlooking the need to isolate co-running applications, leading to significant performance interference when applications share remote memory.

In this chapter, we first perform a detailed performance study to understand the sources of this performance interference, and quantify their performance impact on co-running applications. Based on these findings, we introduce Canvas, an isolation mechanism for the OS kernel swap system that holistically isolates swap resources (*e.g.*, swap partition, swap cache, prefetcher, and RDMA bandwidth) for each application. Furthermore, because swap isolation segregates each application's access patterns and needs, it enables the OS kernel to capture per-application semantics and adaptively optimize the swap system for optimal efficiency. Specifically, Canvas incorporates three such optimizations: (1) adaptive swap entry allocation, (2) semantics-aware prefetching, and (3) two-dimensional RDMA scheduling. With its isolation mechanism, Canvas minimizes performance interference between co-running applications; with its adaptive optimizations, Canvas further enhances performance for each individual application beyond the isolation benefits.

5.1 Introduction

A typical swap system in the OS uses a *swap partition* and *swap cache* for applications to swap data between memory and external storage. The swap partition is a storage-backed swap space. The swap cache is an intermediate buffer between the *local memory* and storage—it caches *unmapped pages* that were just swapped in or are about to be swapped out. Upon a page fault, the OS looks up the swap cache; a cache miss would trigger a *demand swap* and a number of *prefetching swaps*. Swaps are served by RDMA and all fetched pages are initially placed in the swap cache. The demand page is then mapped to a virtual page and moved out of the swap cache, completing the fault handling process.

Problems. Current swap systems run multiple applications over shared swap resources (*i.e.*, swap partition, RDMA, *etc.*). This design works for *disk-based swapping* where disk access is slow—each application can allow only a tiny number of pages to be swapped to maintain an acceptable overhead. This assumption, however, no longer holds under far memory because an application can place more data in far memory than local memory and yet still be efficient, thanks to RDMA's low latency and high bandwidth.

As such, applications have orders-of-magnitude more swap requests under far memory than disks. Millions of swap requests from different applications go through the same shared data path in a short period of time, leading to *severe performance interference*. Our experiments show that, with the same amounts of CPU and local-memory resources, co-running applications leads up to a 6× slowdown, an overhead unacceptable for any real-world deployment.

State of the Art. Interference is a known problem in datacenter applications and a large body of work exists on isolation of CPU [28, 50, 141], I/O [91, 234], network bandwidth [25, 83, 113, 173, 203, 232] and processing [122]. Most of these techniques build on Linux's `cgroup` mechanism, which focuses on isolation of traditional resources such as CPU and memory, *not* swap resources such as remote memory usage and RDMA. Prior swap optimizations such as Infiniswap [89] and Fastswap [12] focus on reducing remote access latency, overlooking the impact of swap interference in realistic settings. Justitia [284] isolates RDMA bandwidth between applications, but does not eliminate other types of interference such as locking and swap cache usage.

Contribution #1: Interference Study (§5.2). We conducted a systematic study with a set of widely-deployed applications on Linux. Our results reveal three major performance problems:

- **Severe lock contention:** Since all applications share a single swap partition, extensive locking is needed for swap entry allocation (needed by every swap-out), reducing throughput and precluding full utilization of RDMA's bandwidth. Our experience shows that in windows of frequent remote accesses, applications can spend **70%** of the windows' time on swap entry allocation.
- **Uncontrolled use of swap resources (*e.g.*, RDMA):** The use of the shared RDMA bandwidth is often dominated by the pages fetched for applications with many threads simultaneously performing frequent remote accesses. For example, aggressively (pre)fetching pages to fulfill one application's needs can disproportionally reduce other applications' bandwidth usage. Further, even within one application, prefetching competes for resources with demand swaps, leading to either prolonged fault handling or delayed prefetching that fails to bring back pages in time.
- **Reduced prefetching effectiveness:** Applications use the same prefetcher, prefetching data based on *low-level (sequential or strided) access patterns* across applications. However, modern applications exhibit far more diverse access patterns, making it hard for prefetching to be effective across the board. For example, co-running Spark and native applications

reduces Leap [158]'s prefetching contribution by **3.19×**.

These results highlight two main problems. First, interference is caused by sharing a combination of swap resources including the swap partition/cache, and RDMA (bandwidth and SRAM on RNIC). Although recent kernel versions added support [105] for charging prefetched pages into `cgroup`, resolving interference requires a *holistic* approach that can isolate all these resources. Furthermore, interference stems not only from resource racing, but also from fundamental limitations with the current design of the swap system. For instance, reducing interference between prefetching and demand swapping requires understanding whether a prefetching request can come back in time. If not, it should be dropped to give resources to demand requests, which are on the critical path. This, in turn, requires a redesign of the kernel's fault handling logic.

Second, cloud applications exhibit highly diverse behaviors and resource profiles. For example, applications with a great number of threads are more sensitive to locking than single-threaded applications. Furthermore, managed applications such as Spark often make heavy use of reference-based data structures while native applications are often dominated by large arrays. The *application-agnostic nature* of the swap system makes it hard for a one-size-fits-all policy (*e.g.*, a global prefetcher) to work well for diverse applications. Effective per-application policies dictates (1) holistic swap isolation and (2) understanding application semantics, which is currently inaccessible in the kernel.

Contribution #2: Holistic Swap Isolation (§5.3). To solve the first problem, we develop Canvas, a *fully-isolated* swap system, which enables each application to have its dedicated swap partition, swap cache, and RDMA usage. In doing so, Canvas can charge each application's `cgroup` for the usage of all kinds of swap resources, preventing certain applications from aggressively invading others' resources.

Contribution #3: Isolation-Enabled Adaptive Optimizations (§5.4). To solve the second problem, we develop a set of adaptive optimizations that can tailor their policies and strategies to application-specific swap behaviors and resource needs. Our adaptive optimizations bring a *further boost* on top of the isolation-provided benefits, making co-running applications even *outperform* their individual runs.

(1) Adaptive Swap Entry Allocation (§5.4.1) Separating swap partitions reduces lock contention at swap entry allocations to a certain degree, but the contention can still be heavy for multi-threaded applications. For example, Spark creates many threads to fully utilize cores and these threads need synchronizations before obtaining swap entries. The synchronization overhead increases dramatically with the number of cores (§5.5.4.1), creating a scalability bottleneck. We develop an adaptive swap entry allocator that dynamically balances between the degree of lock contention (*i.e.*, time) and the amount of swap space needed (*i.e.*, space) based on each application's memory behaviors.

(2) Adaptive Two-tier Prefetching (§5.4.2) Current kernel prefetchers build on low-level access patterns (*e.g.*, sequential or strided). Although such patterns are useful for applications with large array usages, many cloud applications are written in high-level, managed languages such as Java or Python; their accesses come from multiple threads or exhibit pointer-chasing behavior as opposed to sequential or strided patterns. As effective prefetching is paramount to remote-memory performance, Canvas employs a two-tier prefetching design. Our *kernel-tier prefetcher* prefetches data for each application into its private swap cache based on low-level patterns. Once this prefetcher cannot effectively prefetch data, Canvas adaptively forwards the faulty address up to the *application tier* via a modified `userfaultfd` interface, enabling customized prefetching logic at the level of reference-based or thread-based access patterns.

(3) Adaptive RDMA Scheduling (§5.4.3) Isolating RDMA bandwidth alone for each application is insufficient. As there could be many more *prefetching* requests than *demand swap requests*, naïvely sending all to RDMA delays demand requests, increasing fault-handling latency. On the other hand, naïvely delaying prefetching requests (as in Fastswap [12]) reduces their *timeliness*, making prefetched pages useless. We built a *two-dimensional* RDMA scheduler, which schedules packets not only between applications but also between prefetching and demand requests for each application.

Results. Our evaluation (§5.5) with a set of 14 widely-deployed applications (including Spark [274], Cassandra [18], Neo4j [180], Memcached [163], XGBoost [47, 48], Snappy [85], *etc.*) demonstrates that Canvas improves the overall application performance by up to **6.2**× (average **3.5**×) and reduces applications' performance variation (*i.e.*, standard deviation) by **7**×, from an overall of **1.72** to **0.23**. Canvas improves the overall RDMA bandwidth utilization by **2.8**× for co-run applications. Canvas is available at `https://github.com/uclasystem/canvas`.

5.2 Motivating Performance Study

To understand the impact of interference, we conducted a study with a set of widely-deployed applications including Apache Spark [274], Neo4j [180], XGBoost [48] (*i.e.*, a popular ML library), Snappy [85] (*i.e.*, Google's fast compressor/decompressor), as well as Memcached [163]. Spark and Neo4j are managed applications running on the JVM, while the other three are native applications. They cover a spectrum of cloud workloads from data storage through analytics to ML. In addition, they include both batch jobs (such as Spark) and latency-sensitive jobs (such as Memcached). Co-running them represents a typical scenario in a modern datacenter where operators fill left-over cores unused by latency-sensitive tasks with batch-processing applications to improve CPU utilization [27]. For example, in a Microsoft Bing cluster, batch jobs are colocated with latency-sensitive services on over 90,000 servers [106]. Google also reported that 60% of machines in their compute cluster co-run at least five jobs [282].

We ran these programs, individually vs. together, on a machine with two Xeon(R) Gold

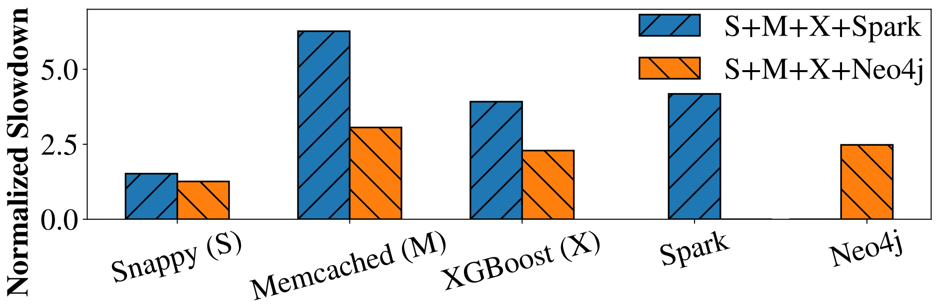


Figure 5.1: Slowdowns of co-running applications compared to running each individually.

6252 processors, running Linux 5.5. Another machine with two Xeon(R) CPU E5-2640 v3 processors and 128GB memory was used for remote memory. Each machine was equipped with a 40 Gbps Mellanox ConnectX-3 InfiniBand adapter and inter-connected by one Mellanox 100 Gbps InfiniBand switch. Using `cgroup`, the same amounts of CPU and local memory resources were given to each application throughout the experiments. RDMA bandwidth was *not* saturated for both application individual runs and co-runs. The amount of local memory configured for each application was 25% of its working set.

Performance Interference and Degradation. To understand the overall performance degradation and how it changes with different applications, we used two managed applications: Spark and Neo4j. Figure 5.1 reports each application's performance degradation when co-running with other applications compared to running alone. The blue/orange bars show the slowdowns when the three native applications co-run with Spark/Neo4j. Clearly, co-running applications significantly reduces each application's performance. We observed an overall **3.9**/**2.2**× slowdown when native applications co-run with Spark/Neo4j. Spark persists a large RDD in memory and keeps swapping in/out different parts of the RDD, while Neo4j is a graph database and holds much of its graph data in local memory and thus does not swap as much as Spark.

Another observation is that the impact of interference differs significantly for different applications. Applications that generate high swap throughputs aggressively invade swap

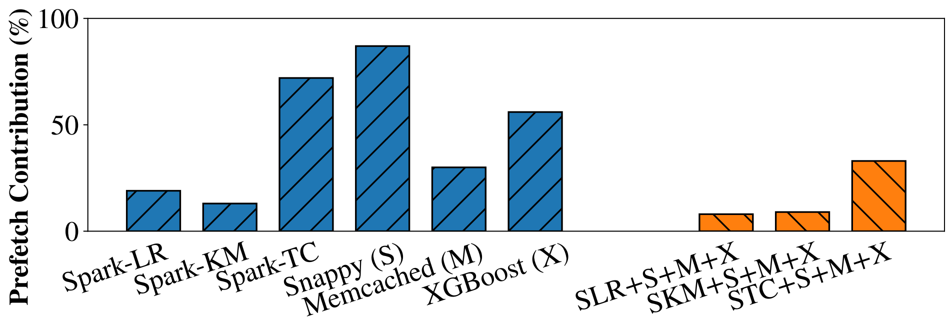
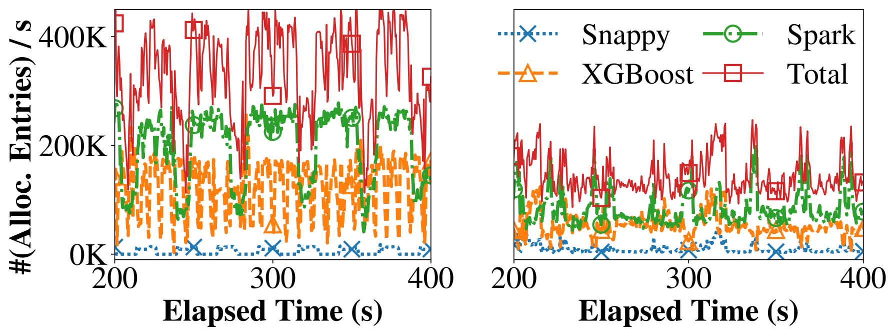


Figure 5.2: Prefetching contribution of Leap: the percentage of page faults served by Leap-prefetched pages (%).

and RDMA resources of other applications. In our experiments, Memcached, XGBoost, and Spark all need frequent swaps. However, Spark runs many more threads (> 90 application and runtime threads) than Memcached (4 threads) and XGBoost (16 threads), resulting in a much higher swap throughput. As such, Spark takes disproportionally more resources, leading to severe degradation for Memcached and XGBoost.

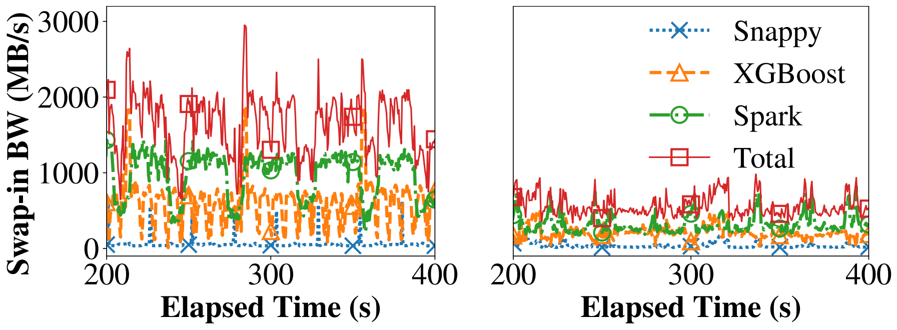
Reduced Prefetching Effectiveness. Sharing the same prefetching policy reduces the prefetching effectiveness when multiple applications co-run. Figure 5.2 reports *prefetching contribution*—the percentage of page faults served by prefetched pages—the higher the better; if a prefetched page is never used, prefetching it would only incur overhead. We used Leap [158] as our prefetcher. The left six bars report such percentages for the applications running individually. When applications co-run, the rightmost three bars report the average percentages across applications. As shown, co-running dramatically reduces the contribution.

Note that Leap [158] uses a majority-vote algorithm to identify patterns across multiple applications. However, when applications that exhibit drastically different behaviors co-run, Leap cannot adapt its prefetching mechanism and policy to each application. Furthermore, Leap is an aggressive prefetcher—even if Leap does not find any pattern, it always prefetches a number of contiguous pages. However, aggressive prefetching for applications such as Spark with garbage collection (GC) is ineffective—*e.g.*, prefetching for a GC thread has zero benefit and only incurs overhead. Detailed evaluation of prefetching can be found in §5.5.4.



(a) Running individually. (b) Co-running.

Figure 5.3: Swap entry allocation throughput when applications run individually (a) and together (b).



(a) Running individually. (b) Co-running.

Figure 5.4: RDMA swap-in bandwidth when applications run individually (a) and together (b).

Lock Contention. We observed severe lock contention in the swap system when applications co-run, particularly at swap entry allocation associated with each swap-out.

We experimented with Spark (Logistic Regression), XGBoost, and Snappy. Our results show that in windows of frequent remote accesses, co-running applications can spend up to **70%** of the window time on obtaining swap entries. Lock contention leads to significantly

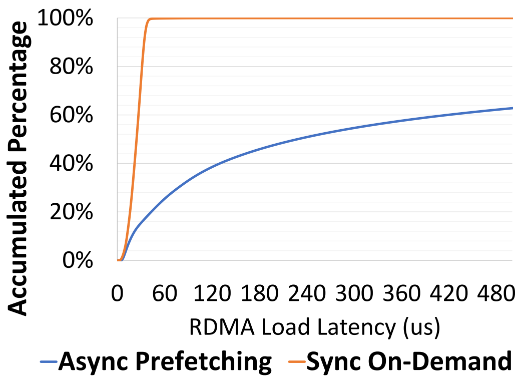


Figure 5.5: Latency of prefetching and on-demand swapping.

reduced swap-entry allocation throughput, reported in Figure 5.3. The total lines in Figure 5.3(a) and (b) show the total throughput (*i.e.*, the sum of each application's allocation throughput). The co-running throughput (b) is drastically reduced compared to the individual run's throughput (a) (*i.e.*, ~450Kps to ~200Kps).

Reduced RDMA Utilization. Figure 5.4 compares the RDMA read bandwidth (for swap-ins) when applications run individually and together. Similarly, the total line represents the sum of each application's RDMA bandwidth. The total RDMA utilization is constantly below ~1000MBps in Figure 5.4(b), which is **3.28×** lower than that in Figure 5.4(a) due to various issues (*e.g.*, locking, reduced prefetching, *etc.*). The RDMA write bandwidth degrades by an overall of **2.80×**.

Demand v.s. Prefetching Interference. Optimizations such as Fastswap [12] improve swap performance by dividing the RDMA queue pairs (QP) into sync and async. The high-priority synchronous QP is used for demand swaps, while the low priority async QP is used for prefetching requests. This separation reduces head-of-line blocking incurred by prefetching. However, when applications co-run, this design adds a delay for prefetching. Figure 5.5 depicts the CDF of the latency of RDMA packets from demand and prefetching requests, when the four applications co-run on Leap. As shown, 99% of the on-demand requests are served within 40µs. However, the latency of 36.9% of prefetching requests is longer than 512µs and it can reach up to 52ms! Long latency renders prefetched pages useless because prefetching is meant to load pages to be used soon. Our profiling shows that among the prefetched pages that are actually accessed by the application, *90% are accessed within 70µs*, indicating that ~70% of the pages prefetched return too late. A late prefetch of a page would subsequently block a demand request of the page when it is accessed by the application. This problem motivates our two-dimensional RDMA scheduling (§5.4.3).

Problem Description	**Performance Impact**	**Canvas's Solution**
Unlimited use of swap RDMA resources	Apps generating higher swap thruput use disproportionately more resources	Holistic isolation of swap system and RDMA isolation and scheduling (§5.3, §5.4.3)
Lock contention at swap entry allocation	Reduced swap-out throughput	(1) Swap parti. isolation (§5.3); (2) adaptive entry alloc. (§5.4.1)
Single low-level prefetcher	Increased fault-handling latency	Two-tier adaptive prefetching (§5.4.2)
prefetching v.s. demand interfere	Increased fault-handling latency	Two-dimensional RDMA scheduling (§5.4.3)

Table 5.1: Summary of major issues and Canvas's solution.

Takeaway. The root cause of performance degradation is that multiple applications, whose resource needs and swap behaviors are widely apart, all run on a global swap system with the same allocator and prefetcher. Table 5.1 summarizes these problems, their performance impact, and our solutions.

5.3 Swap System Isolation

Canvas extends `cgroup` for users to specify size constraints for swap partition, swap cache, and RDMA bandwidth. We discuss the kernel support to enforce these new constraints, laying a foundation for adaptive optimizations in §5.4.

Swap Partition Isolation. In Linux, remote memory is managed via a swap partition interface, shared by all applications. If there are multiple available swap partitions, they are used in a *sequential manner* according to their priorities. As a result, data of different applications are mixed and stored in arbitrary locations.

Canvas separates remote memory of each `cgroup` to isolate capacity and performance. The user creates a `cgroup` to set a size limit of remote memory for an application. Canvas allocates remote memory in a demand-driven manner—upon a pressure in local memory, Canvas allocates remote memory and registers it as a RDMA buffer. Canvas enables per-`cgroup` swap partitions by creating a swap partition interface and attaching it to each `cgroup`. For each `cgroup`, a separate swap-entry manager is used for allocating and freeing swap entries. Swap entry allocation can now be charged to the `cgroup`, which controls how much remote memory each application can use. Our adaptive swap entry allocation algorithm is discussed in §5.4.1.

Canvas explicitly enables a private swap cache for each `cgroup` (a default value of 32MB), whose size is charged to the *memory budget* specified in the `cgroup`. As a result, the size of an application's swap cache changes in response to its own memory usage, without affecting other applications.

For each demand swap-in, Canvas first checks the `mapcount` of the page, which indicates how many processes this page has been mapped to before. If the page belongs only to one process, it is placed in its private swap cache. Otherwise, it has to be placed in a global swap cache (discussed shortly). To release pages (*e.g.*, when the application's working set increases, pushing the boundary of the swap cache), Canvas scans the swap cache's page list, releasing a batch of pages to shrink the cache.

RDMA Bandwidth Isolation. For each `cgroup`, Canvas isolates RDMA bandwidth with a set of *virtual* RDMA queue pairs (VQPs) and a centralized packet scheduler. Users can set the swap-in/swap-out RDMA bandwidth of a `cgroup` with our extended interface. Our RDMA scheduler works in two dimensions. The *first dimension* schedules packets across applications, while the *second dimension* schedules on a per-application basis—each cgroup has its *sub-scheduler* that schedules packets that belong to the cgroup between demand swapping and prefetching.

VQPs are high-level interfaces, implemented with lock-free linked lists. Each cgroup pushes its requests to the head of its VQP, while the scheduler pops requests from their tails. At the low level, our scheduler maintains three physical queue pairs (PQP) per core, for *demand swap-in*, *prefetching*, and *swap-out*, respectively. The scheduler polls all VQPs and forwards packets to the corresponding PQPs, using a *two-dimensional* scheduling algorithm (see §5.4.3).

Handling of Shared Pages. Processes can share pages due to shared libraries or memory regions. These pages cannot go to any private swap cache. Canvas maintains a global swap partition and cache for shared pages. When a page is evicted and ummapped, Canvas checks its `mapcount` and adds it to the global swap cache if the page is shared between different processes. All pages in the global swap cache will be eventually swapped out to the global partition using the original lock-based allocation algorithm. Conversely, pages swapped in (and prefetched) from the global swap partition are all placed into the global swap cache. For typical cloud applications such as Spark, Cassandra and Neo4j, the number of shared pages is much smaller than process-private pages, using locks in a normal way would not incur a large overhead. We cannot charge applications' cgroups for pages in the global swap cache, because which process(es) share these pages is unknown before they get mapped into processes' address spaces. Canvas allows users to create a special cgroup, named cgroup-shared, to limit the size of the global swap cache/partition.

One limitation of our cgroup-based approach is that cgroup can only partition resources statically while applications' resource usage may change from time to time and static partitioning could lead to resource underutilization. However, the focus of this chapter is to ensure isolation and future work could incorporate max-min fair allocation to improve resource utilization.

5.4 Isolation-Enabled Swap Optimizations

On top of the isolated swap system, we develop three optimizations, which dynamically adapt their strategies to each application's resource patterns and semantics.

5.4.1 Adaptive Swap Entry Allocation

As discussed in §5.2, swap entry allocation suffers from severe lock contention under frequent remote accesses—allocation is needed at every swap-out. To further motivate, we use a simple experiment by running Memcached alone on remote memory with different core numbers. As the number of cores increases, the average entry allocation time grows super-linearly—it grows from 10µs under 16 cores quickly to 130µs under 48 cores due to increased lock contention (see Figure 5.12). Creating a per-application swap partition mitigates the problem to a certain degree. However, applications like Spark run more than 90 threads; frequent swaps in these threads can still incur significant locking overhead.

To further reduce contention, we develop a novel swap entry allocator that adapts allocation strategies in response to each application's own memory access/usage. Our first idea is to enable a *one-to-one* mapping between pages and swap entries. At the first time a page is swapped out, we allocate a new swap entry using the original (lock-protected) algorithm. Once the entry is allocated, Canvas writes the entry ID into the page metadata (*i.e.*, `struct page`). This ID remains on the page throughout its life span. As a result, subsequent swap-outs of the page can write data directly into the entry corresponding to this ID. We pay the locking overhead *only once* for each page at its first swap-out.

This approach requires a swap entry to be reserved for each page. For example, if the local memory size is S and the remote memory allocation is 3S, with one-to-one mapping the remote memory allocation would be 4S (*i.e.*, each page residing in local memory also has a remote page, resulting in a 33% overhead). However, this overhead may not be necessary. For example, modern applications exhibit strong epochal behaviors. Under the original allocator, swap entries for pages accessed in one epoch can be reused for those in another epoch. Under this approach, however, all pages in all epochs must have their dedicated swap entries throughout the execution, which can lead to an order-of-magnitude increase in remote memory usage.

Our key insight is: we should trade off *space for time* if an application has much available swap space, but *time for space* when its space limit is about to be reached. As such, when the remote memory usage is about to reach the limit specified in `cgroup` (*i.e.*, 75% in our experiments), Canvas starts removing reservations to save space. The next question is which pages we should consider first as our candidates for reservation removal. Our idea is that we should first consider "hot pages" that always stay in local memory and are rarely swapped. This is because hot pages (*i.e.*, data on such pages are frequently accessed) are likely to stay in local memory for a long time; hence, locking overhead is less relevant for them. On the contrary, "cold" pages whose accesses are *spotty* are more likely to be swapped in/out and hence swap efficiency is critical. Here "hot" and "cold" pages are relatively defined as they are specific to execution stages—a cold page swapped out in a previous stage can be swapped in and become hot in a new stage.

To this end, we develop an *adaptive allocator*. Canvas starts an execution by reserving swap entries for *all* pages to minimize lock contention. Reservation removal begins when remote-memory pressure is detected. Canvas adaptively removes reservations for hot pages. We detect hot pages *for each application* by periodically scanning the application's *LRU active list*—pages recently accessed are close to the head of the active list. Each scan identifies a set of pages from the head of the list; a page is considered "hot" if it appears in a consecutive number of sets recently identified.

Removing the reservation for a hot page can be done by (1) removing the entry ID from the page metadata and (2) freeing its reserved swap entry in remote memory, adding the

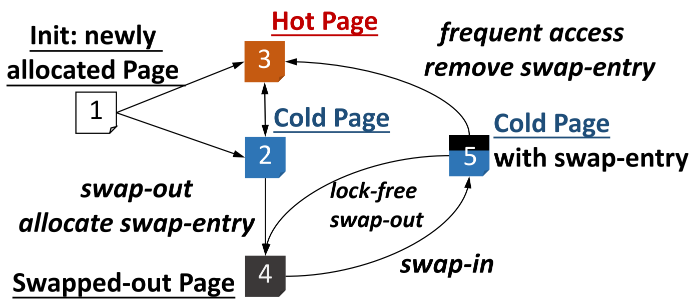

Hot Page
Init: newly allocated Page
frequent access remove swap-entry
1
3
Cold Page
2
Cold Page with swap-entry
5
swap-out allocate swap-entry
lock-free swap-out
swap-in
Swapped-out Page
4

Figure 5.6: FSM describing our page management when remote-memory pressure is detected.

entry back to the free list. Once a hot page becomes cold and gets evicted, it does not have a reservation any more, and hence, it goes through the original (lock-protected) allocation algorithm to obtain an entry. In this case, the page receives a new swap entry and remembers this new ID in its metadata.

Figure 5.6 shows the page state machine, which describes the page handling logic. A cold page (to be evicted) can be in one of the two states: state 2 and state 5. A page comes to state 2 if it is (1) a brand new page that has never been swapped out or (2) previously a hot page but has not been accessed for long. Once it reaches state 2, the page does not have a reserved swap entry ID and hence, swapping out this page goes through the normal allocation path. In the case of swap-in (state 5), the swap entry ID is already remembered on the page. The next swap-out will directly use this entry and be lock-free. If the page becomes hot (from state 5 to 3), Canvas removes the entry ID and releases the entry reservation. The entry is then added back to the free list.

Performance Analysis. To understand the performance of the adaptive entry allocation algorithm, let us consider the following two scenarios. In the first scenario, the application performs uniformly random accesses. As a result, Canvas cannot clearly distinguish hot/cold pages, and thus randomly cancels their reservations. However, due to the random process, when a page is swapped out, it has a certain probability of still possessing a reserved swap entry (depending on the ratio of remaining reservations) and hence Canvas can still improve the allocation performance.

In the second scenario, the application follows a repetitive pattern of accessing a page a few times (making it hot) and then moving on to accessing another page; it will not come back to the page in a long while. Under our allocation algorithm, every page will be identified as a hot page, leading to the cancellation of its reservation. However, each page will be swapped out when it is cold enough; at each swap-out, the page has to go through the original allocation algorithm. This is the worst-case scenario, and even in this case, Canvas has the same (worst-case) performance as the original Linux allocator, which allocates an entry at each swap-out.

Some of the recent patches submitted to the Linux community also attempt to reduce lock contention for swap entry allocation.

5.4.2 Two-Tier Adaptive Prefetching

Problems with Current Prefetchers. Current prefetchers all focus on low-level (streaming or strided) access patterns. While such patterns exist widely in native array-based programs, applications written in high-level languages such as Python and Java are dominated by reference-based data structures—operations over such data structures involve large amounts of pointer chasing, making it hard for current prefetchers to identify clear patterns.

Furthermore, cloud applications such as Spark are heavily multi-threaded. Modern language runtimes, such as the JVM, run an additional set of auxiliary threads, *e.g.*, for GC or JIT compilation. How these user-level threads map to kernel threads is often implemented differently in different runtimes. Consequently, kernel prefetchers such as Leap [158] cannot distinguish patterns from different threads.

To develop an adaptive prefetcher, Canvas employs a two-tier design, illustrated in Figure 5.7. At the low (kernel) tier, Canvas uses an existing kernel prefetcher that prefetches data for each application into its own private swap cache (unless data comes from the global swap partition). A kernel prefetcher is extremely efficient and can already cover a range of (array-based) applications. For applications whose accesses are too complex for the kernel prefetcher to handle, we forward the addresses up to the application level, letting the application/runtime analyze semantic access patterns at the level of threads, references, arrays, *etc.*

Prefetching Logic. In Canvas, we adopt the sync/async separation design in Fastswap [12], which prevents head-of-line blocking. As stated earlier, we use three PQPs per core, one for swap-out, one for (sync) demand swap-in, and one for (async) prefetching. Canvas polls for completions of critical (demand) operations, while configuring *interrupt completions* for asynchronous prefetches.

Canvas determines whether to use an application-tier prefetcher based on *how successful kernel-tier prefetching is.* If the number of pages prefetched for an application is lower than a threshold at the most recent N (=3 in our evaluation) faults consecutively, Canvas starts forwarding the faulting addresses up to the application-tier prefetcher (discussed shortly) although the kernel-tier prefetcher is still used as the first-line prefetcher.

Canvas stops forwarding whenever the kernel-tier prefetcher becomes effective again. Our key insight is: the kernel-tier prefetcher is efficient without needing additional compute resources (as it uses the same core as the faulting thread), while the application-tier prefetcher needs extra compute resources to run. As such, we disable application-tier prefetchers as long as the kernel-tier prefetcher is effective. To pass a faulting address to the application, we modify the kernel's `userfaultfd` interface, allowing applications to handle faults at the user space. Our modification makes the kernel forward the faulting address only if the kernel's prefetcher continuously fails to prefetch pages.

Runtime Support for Application-tier Prefetching. A major challenge is how to develop application-tier prefetchers. On the one hand, application-tier prefetchers should

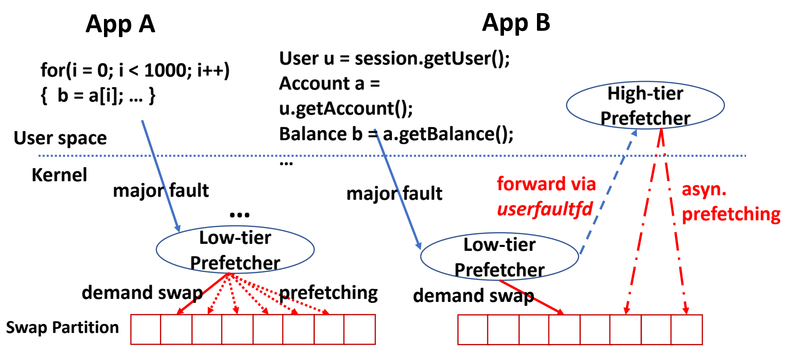


Figure 5.7: Canvas's two-tier prefetcher: App A is an array-based program while B is a modern web application that uses reference-based data structures. The low-tier prefetcher successfully prefetches pages for A, but not for B. Hence, Canvas forwards the addresses up to B's high-tier prefetcher.

conduct prefetching based on *application semantics*, of which the kernel is unaware. On the other hand, application developers may not be familiar with a low-level activity like prefetching; understanding memory access patterns and developing prefetchers can be a daunting task for them.

Our insight is: applications that benefit from application-tier prefetching are mostly written in high-level languages and run on a managed runtime such as the JVM. Inspired by previous work on using language runtime to solve memory efficiency problems for data analytics applications [155, 177, 181–183], Canvas currently supports application-tier prefetching for the JVM as a platform. However its support could be easily extended to other managed runtimes for high-level languages like Go and C#. Leveraging language runtime solves both problems discussed above—it has access to semantic information such as how objects are connected and the number of application threads; furthermore, the burden of developing an application-tier prefetcher is shifted from application developers to runtime developers. Thus, it is not necessary to supply a custom application-tier prefetcher per application, but define it once for each language runtime.

In this work, we develop an application-tier prefetcher in Oracle's OpenJDK as a proofof-concept. It works for all (Java, Scala, Python, *etc.*) programs that run on the JVM. Our JVM-based prefetcher considers two *semantic patterns*: (1) *reference-based* (*i.e.*, accessing an object brings in pages containing objects referenced by this object) and (2) *thread-based* (*i.e.*, accesses from different application threads are separately analyzed to find patterns).

For (1), we modify the JVM to add support that can quickly find, from a faulting address, *the object* in which the address falls. We use write barrier, a piece of code instrumented by the JVM at each object field write, as well as the garbage collector to record references between pages. For example, for each write of the form `a.f=b`, if the objects referenced by a and b are on different page groups, we record an edge on a *summary graph* where each node represents a consecutive group of pages and each edge represents references between groups. During prefetching, we traverse the graph from the node that represents the accessed page and prefetch pages that can be reached within 3 hops. The traversal does not follow cycles and its overhead is negligible. This approach is suitable for applications that store a large amount of data in memory, such as Spark and Cassandra.

For (2), we leverage the JVM's user-kernel thread map. For each faulting address, Canvas additionally forwards the thread information (*i.e.*, pid) to the JVM, which consults the map to filter out non-application (*e.g.*, GC, compilation, *etc.*) threads and segregate addresses based on Java threads (as opposed to kernel threads). Segregated addresses allow us to analyze (sequential/strided) patterns on a per-thread basis (using Leap's majority-vote algorithm [158]). Once patterns are found, the prefetcher sends the prefetching requests to the kernel via `async_prefetch`.

For native programs that directly use kernel threads (*e.g.*, `pthread`), the thread information is straightforward and immediately visible to Canvas. We can easily segregate addresses accessed from different threads and analyze patterns based upon addresses from each individual thread.

Policy. To improve effectiveness, the JVM uses a search tree to record information about large arrays. Upon the allocation of an array whose size exceeds a threshold (*i.e.*, 1MB in our experiments), the JVM records its starting address and size into the tree. The JVM runs a daemon prefetching thread. Once receiving a sequence of faulting addresses, we determine which semantic pattern to use based on *how many application threads are running* and *whether the faulting addresses fall into a large array.* If there are many threads and the faulting addresses fall into arrays, the JVM uses (2) to find per-thread patterns. If either condition does not hold, the JVM uses (1) to prefetch based on references. For native applications, we only enable (2), as we observed that our native programs do not use many deep data structures.

5.4.3 Two-Dimensional RDMA Scheduling

To provide predictable performance for applications sharing RDMA resources, our RDMA scheduling algorithm should provide four properties: (1) weighted fair bandwidth sharing [32, 67] across applications; (2) high overall utilization; (3) treating demand and prefetching requests with different priorities; and (4) timely handling of prefetching requests.

Canvas performs two-dimensional scheduling by extending existing techniques. Canvas uses max-min fair scheduling to assign bandwidth across applications, and priority-based scheduling with *timeliness* to schedule prefetching and demand requests within each application. Although these scheduling techniques are not new themselves, Canvas combines them in a unique way to solve the interference problem. Canvas maintains three PQPs on each core, respectively, for swap-outs, demand swap-ins, and prefetching swap-ins. Swap-outs are only subject to fair scheduling while swap-ins are subject to both fair and priority-based scheduling.

Vertical: Fair Scheduling. Under max-min fairness, each application receives a fair share of bandwidth. If there is extra bandwidth, we give it to the applications in the reverse order of their bandwidth demand until bandwidth is saturated. The high overall utilization of bandwidth is achieved by redistributing unconsumed bandwidth proportionally to the weights of unsatisfied applications. Canvas implements weighted fair queuing with virtual clock [67, 195, 279].

Horizontal: Priority Scheduling with Timeliness. Within each `cgroup`, Canvas schedules demand requests with a higher priority than prefetching requests. However, this could lead to long latency for prefetching requests. To bound the latency of prefetching, our scheduler employs a history-based heuristic algorithm to identify and drop outdated prefetching requests. In particular, Canvas maintains the *timeliness distribution* of prefetched pages per `cgroup`. Timeliness is a metric that measures the time between a page being prefetched and accessed. We attach a timestamp to each request when pushing it into a VQP. The scheduler maintains packets statistics on-the-fly to estimate the round-trip latency and arrival time of each prefetching request. Requests are dropped if the estimated arrival time exceeds the estimated timeliness threshold.

Special care must be taken to drop prefetching requests. Before issuing a prefetching request, the kernel creates a page in the swap cache and sets up its corresponding PTE. The page is left in a *locked* state until its data comes back. However, a thread that accesses an address falling into the page may find this locked page in the swap cache and block on it. Dropping prefetching requests may cause the thread to hang. To solve the problem, we detect threads that block on prefetching requests for too long and generate new *demand requests* for them.

We rely on a per-entry timestamp to efficiently detect threads that block on prefetching requests. In Canvas, we attach a timestamp field to the swap entry metadata. Canvas's scheduler records the timestamp every time it enqueues a prefetching request into VQP. If another thread faults on the same page later, it will retrieve the same swap entry from the PTE. If the swap entry contains a timestamp, the faulting thread knows that a prefetching request has already been issued. Next, the faulting thread calculates the time elapsed since the timestamp, and compares it with a timeout threshold (maintained by the RDMA scheduler based on page-fetching latencies). If it exceeds the timeout threshold, the faulting thread drops the prefetching request. The drop operation is elaborated below:

Before issuing each (demand or prefetching) request, the kernel first allocates a physical page in the swap cache and locks the page until the request returns. Upon the return of the data, the data is written into the page; the page is unlocked and mapped into the page table. In order to safely drop a request, we add another field *valid* in the swap entry metadata, indicating whether the prefetching request on the go is valid. Once a faulting thread identifies a delayed prefetching request (by using the timestamp as discussed above), it sets the *valid* field in the swap entry to `false` and then creates a new physical page in the swap cache. The thread goes ahead and issues another (demand) I/O request based on this new page. When the delayed prefetching request returns, it checks the *valid* field and discards itself once it sees the false value. The field is then set back to `true`.

When a demand request is issued, Canvas clears the timestamp field in its corresponding swap entry. If a thread faults on the same page, it will block on the request instead of issuing a new one due to the empty timestamp (indicating that the request on the go is a demand one).

5.5 Evaluation

We implemented the isolation support and adaptive optimizations of Canvas in Linux 5.5, while the application-tier prefetcher was implemented in OpenJDK 12.

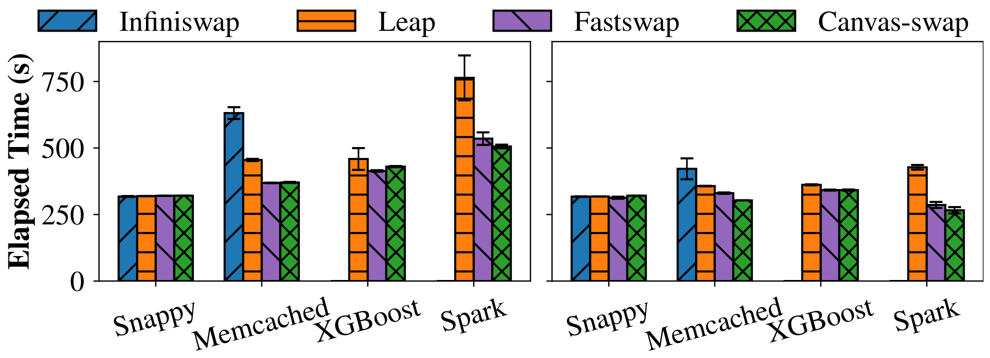
Setup. We included a variety of cloud applications in our experiments, including managed (Java) applications such as Spark [274], Cassandra [18] (a NoSQL database), Neo4j [180] (a graph database), as well as three native applications: XGBoost [48], Snappy [85], and

Application	**Workload**	**Dataset**	**Size** / $(\|E\|, \|V\|)$
Managed			
Cassandra	5M read, 5M insert	YCSB[56]	10M records
Neo4j	PageRank	Baidu[126]	(17M, 2M)
Spark	PageRank (**SPR**)	Wikipedia[126]	(57M, 1.5M)
	KMeans (**SKM**)	Wikipedia[126]	188M points
	Logistic Regression (**SLR**)	Wikipedia[126]	188M points
	Skewed Groupby (**SSG**)	synthetic	256K records
	Triangle Counting (**STC**)	synthetic	(1.5M, 384K)
MLlib	Bayes Classifiers (**MBC**)	KDD [43]	1.5M instances
GraphX	Connected Components (**GCC**)	Wikipedia[126]	(188M, 9M)
	PageRank (**GPR**)	Wikipedia[126]	(188M, 9M)
	Single Src. Shortest Path (**GSP**)	synthetic	2M vertices
Native			
XGBoost	Binary Classification	HIGGS[24]	22M instances
Snappy	Compression	enwik9 [156]	16GB
Memcached	45M gets, 5M sets	YCSB[56]	10M records

Table 5.2: Programs and their workloads.

Memcached [163]. Spark, Cassandra, Neo4j, Memcached, and XGBoost are multi-threaded while Snappy is single-threaded. The Spark applications span popular libraries such as GraphX and MLlib.

We co-ran different combinations of programs. The same application in different combinations receives the same amount of local (CPU and memory) resources. To simplify performance analysis, we let each combination of applications co-run contain one managed (Spark, Cassandra, or Neo4j) application and the three native programs, which consume less resources. These experiments were conducted on two machines, one used to execute applications and a second to provide remote memory. The configurations of these machines was reported earlier in §5.2. We carefully configured Linux with the following configuration to achieve the best performance for Linux: (1) SSD-like swap model, (2) per-VMA prefetching



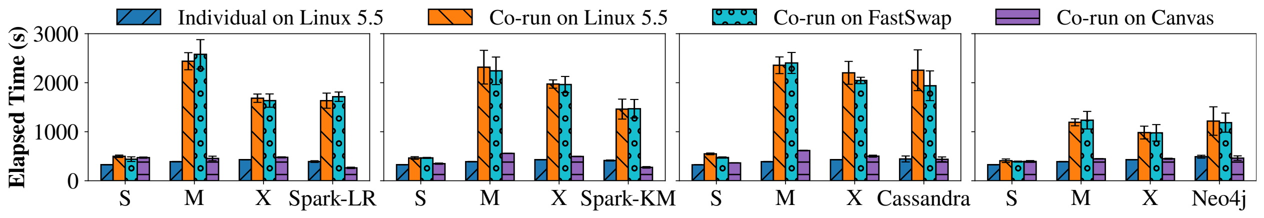
(a) 25% local memory. (b) 50% local memory.

Figure 5.8: Performance of different swap systems.

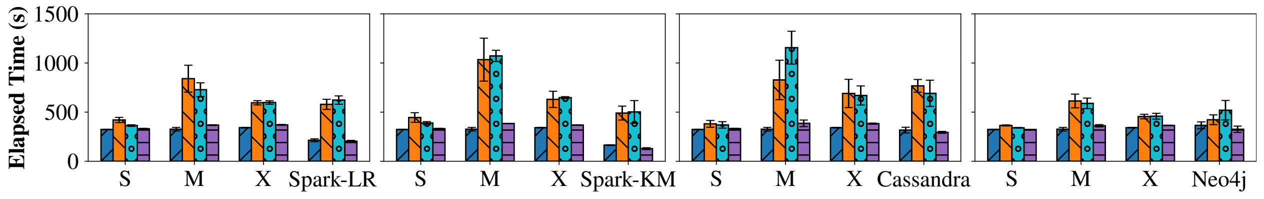
policy, and (3) cluster-based swap entry allocation. We disabled hyper-threads, CPU C-states, dynamic CPU frequency scaling, transparent huge pages, and the kernel's mitigation for speculation attacks.

For each combination, we limited the amounts of CPU resources for the managed application, XGBoost, Memcached, and Snappy to be 24, 16, 4, and 1 core(s). For local memory, we used two ratios: 50% and 25%, meaning each application has 50/25% of its working set locally. When using Canvas, we additionally limited the sizes of swap partitions in such a way that for each application the total size of its swap partition and assigned local memory is slightly larger than its working set. In doing so, each application has just enough (local and remote) memory to run and reservation cancellation (§5.4.1) is triggered in all executions.

The swap cache size for each application starts at 32MB and changes dynamically. The global swap cache size (configured by `cgroup-share`) was also set to 32MB. Canvas uses max-min fair scheduling to assign bandwidth across applications, and their initial weights are proportional to their swap partition assignments. We ran each application 10 times. Their average execution times (with error bars) are reported in all experiments throughput this section.



(a) 25% local memory.



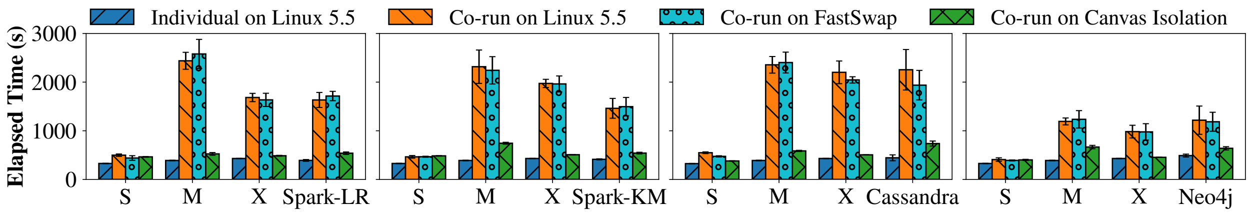
(b) 50% local memory.

Figure 5.9: Performance of each program under 25% and 50% local memory when the three native programs, Snappy (S), Memcached (M), and XGBoost (X), co-run with a managed application. Canvas ran with all optimizations enabled.

5.5.1 Basic Swap Systems

We used Fastswap [12] as our underlying swap system, with a small amount of code changes to port Fastswap (originally built against Linux 4.11) to Linux 5.5. We first compared the performance of each individual application running on basic swap systems including Infiniswap [89], Infiniswap with Leap [158], the original Fastswap [12], and Canvas's ported Fastswap (without isolation and optimizations). We could not run LegoOS [224] as it does not support network-related system calls, which are required for applications such as Spark. LegoOS implements swaps with RPCs as opposed to paging, but our idea (*i.e.*, isolation and adaptive swapping) is applicable to this approach as well.

We ran Infiniswap and Leap on Linux 4.4, and Fastswap on Linux 4.11 to align with their original setup. The results are reported in Figure 5.8. Infiniswap hung on XGBoost and Spark, and its corresponding bars are thus not reported. Since Canvas-swap was built off Fastswap, they have similar performance.



(a) Co-run with Spark-LR. (b) Co-run with Spark-KM. (c) Co-run with Cassandra. (d) Co-run with Neo4j.

Figure 5.10: Performance of native applications co-run with different managed applications under 25% local memory; for Canvas, only isolation was enabled (*i.e.*, without adaptive optimizations).

5.5.2 Overall Performance

Next, we demonstrate the overall performance when applications co-run together under Canvas. Each experiment ran the same set of three native programs with one managed application: Spark-LR, Spark-KM, Cassandra, or Neo4j. The results for the 25% and 50% local memory configurations are reported in Figure 5.9(a) and (b), respectively.

The four bars in each group represent an application's performance when running alone on Linux 5.5, co-running with other applications on Linux 5.5, co-running on the original Fastswap, and co-running on Canvas (with all optimizations enabled). Across all experiments, Canvas improves applications' co-run performance by up to **6.2**× (average **3.5**×) and up to **3.8**× (average **1.9**×) under the two memory configurations. Canvas enables Spark and Neo4j to even outperform their individual runs due to the optimizations that could also improve single-application performance.

5.5.3 Isolation Reduces Degradation and Variation

This experiment measures the effectiveness of isolation alone. We used a variant of Canvas with the isolated swap system and RDMA bandwidth (*i.e.*, vertical scheduling between applications) but without our swap-entry optimization, two-tier prefetcher, and horizontal

Table 5.3: Performance variations of three native applications when co-running with each of the 11 managed applications under 25% local memory (Canvas / Linux 5.5 / Fastswap).

Program	**Mean**			**Min**			**Max**			**σ**		
Snappy	**1.07**	1.28	1.23	**1.03**	1.10	1.08	**1.23**	1.69	1.46	**0.07**	0.20	0.14
Memcached	**1.45**	3.24	3.76	**1.30**	1.48	2.05	**1.91**	6.05	8.17	**0.20**	1.82	2.14
XGBoost	**1.05**	3.17	2.81	**1.01**	1.38	1.91	**1.13**	6.13	4.76	**0.04**	1.59	1.11
Overall	**1.21**	2.56	2.60	**1.01**	1.10	1.08	**1.91**	6.13	8.17	**0.23**	1.64	1.72

RDMA scheduling.

Degradation Reduction. We ran the same set of experiments under 25% local memory. As shown in Figure 5.10, isolation reduces the running time by up to 5.2×, with an average of 2.5×. Isolation is particularly useful for applications that do not have many threads but need to frequently access remote memory, such as Memcached, which has 4 threads and cannot compete for resources with managed applications such as Spark and Cassandra, which have more than 90 (application and runtime) threads. As such, its performance is improved by **3.3**× with dedicated swap resources. Isolation improves the average RDMA utilization by 2.8× from 692MB/s to 1908MB/s, making the peak bandwidth reach 4494MB/s.

Variation Reduction. One significant impact of interference is performance variation—the same application has drastically different performance when co-running with different applications (as shown in Figure 5.1). To demonstrate our benefits, we co-ran the three native applications with each of the eleven managed applications listed in Table 5.2, which cover a wide spectrum of computation and memory access behaviors. Table 5.3 reports various statistics of their performance including the mean, minimum, maximum, and standard deviation of their slowdowns (compared to their individual runs). Clearly, the performance of the three programs is much more stable (indicated by a small σ) under Canvas than Linux—variations are reduced by 7× overall.

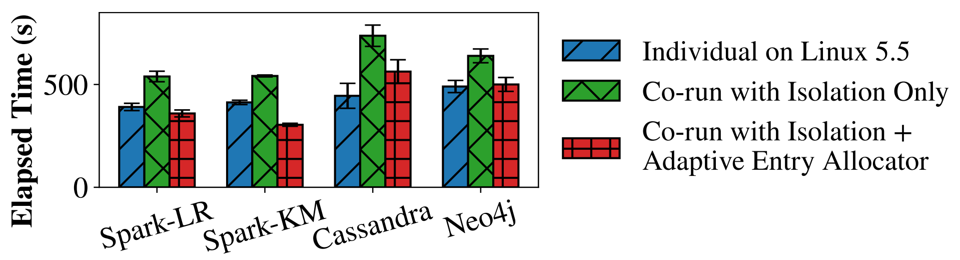


Figure 5.11: Benefit of adaptive swap entry allocation. Compared are the times of the application running individually on Linux 5.5, co-running on Canvas with adaptive entry allocation disabled, and enabled.

5.5.4 Effectiveness of Adaptive Optimizations

This subsection evaluates the benefit of each swap optimization *on top of the isolated swap system* by turning it on/off.

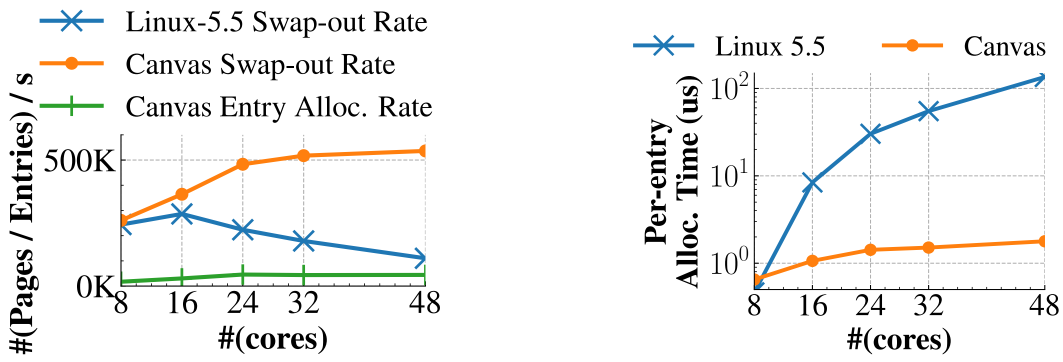
5.5.4.1 Adaptive Swap Entry Allocator

Isolation already reduces lock contention at swap entry allocation because each process has its own swap entry manager. However, for multi-threaded applications such as Spark and Cassandra, their processing threads still have to go through the locking process. In this subsection, we focus on managed applications due to their extensive use of threads. Figure 5.11 shows the performance of Spark LR, Spark KM, Cassandra, and Neo4j when they each co-run with the other three native programs. On average, our adaptive allocation enables an *additional* boost of 1.50× for Spark LR, 1.77× for Spark KM, 1.31× for Cassandra, and 1.28× for Neo4j.

Table 5.4 reports the swap-out throughput when the native applications co-run with Spark. As shown, isolation improves the throughput by 1.67× while adaptive allocation provides an additional boost of 1.51×. This benefit is obtained after applying all optimizations in Linux 5.5.

Table 5.4: Swap-out throughput w/ and w/o adaptive swap-entry allocation when native programs co-run with Spark.

Throughput (KPages/s)	Linux 5.5	Canvas w/o adaptive allocation	Canvas w/ adaptive allocation
Avg. Spark apps	98	164	**295**
Avg. all apps	185	309	**468**



(a) Swap-out and entry alloc rates.

(b) Per-entry alloc time.

Figure 5.12: Entry allocation comparison between the allocation algorithm in Canvas and Linux 5.5 for Memcached under 25% local memory. The Y-axis in (b) is log-scaled.

Effectiveness of Entry Reservation. We compared our adaptive allocation algorithm with the original allocator in Linux 5.5 by running Memcached with varying (8 – 48) cores under 25% local memory. As shown in Figure 5.12(a), for Canvas, (1) the swap-out rate increases with the core number (showing good scalability) and (2) the swap entry allocation rate remains low. This is due to Canvas's entry reservation algorithm that effectively reuses a significant number of swap entries for page swap-outs. On the contrary, in Linux, the swap-out rate (which is the same as its entry allocation rate) decreases when more cores are used. This is because each entry allocation takes significantly longer, reducing the swap-out throughput. A comparison of per-entry allocation time can be seen in Figure 5.12(b). Interested readers can refer to our original paper [260], which also compares the allocation algorithm between Canvas, Linux 5.5, and Linux 5.14.

5.5.4.2 Prefetching Effectiveness

Our baseline is the kernel's default prefetcher on the isolated swap system with adaptive swap allocator *enabled*. Since application-tier prefetching is designed primarily for high-level languages, here we focus on managed programs.

Time. We compare the running time for three Spark applications LR, KM, TC, and Neo4j, between the kernel's prefetcher over Canvas's isolated swap system and Canvas's two-tier prefetcher, when each managed application co-runs with the three native applications under the 25% local memory configuration. Application-tier prefetching brings **33%**, **17%**, **19%**, and **8%** additional performance benefits on top of the kernel prefetching with the isolated swap system. All the four managed applications benefit from the thread-level pattern analysis while the managed applications have seen 5-9% contributions from using the reference-based pattern. The thread-level pattern analysis we added for native programs brings a 5% and 11% improvement for Memcached and XGBoost.

We have also run Leap [158], a prefetcher that aggressively prefetches a number of contiguous pages if it cannot find any pattern. This approach may work for native applications because these applications access arrays; hence, the contiguous pages aggressively prefetched are likely to be useful for array accesses. However, it works poorly for high-level language applications such as Spark and Neo4j, which use deep data structures and run graph-traversal GC tasks (which exhibit neither sequential nor strided patterns). Aggressively prefetching useless pages wastes the RDMA bandwidth and the swap cache. Leap slows down our managed applications by 1.4×, compared to the kernel's default prefetcher.

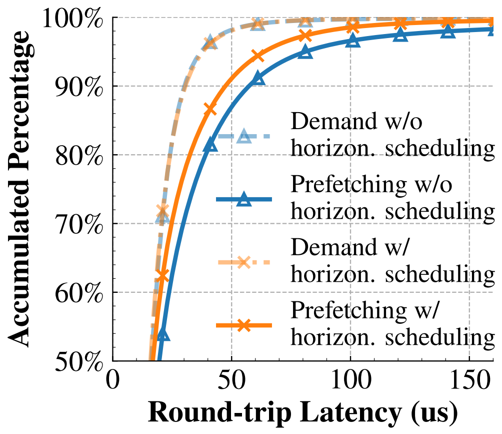
Prefetching Contribution and Accuracy. Table 5.5 compares prefetching *contribution* and *accuracy* for the four managed applications when each of them co-runs with the same three native applications. Contribution is defined as a ratio between the number of page faults hitting on the swap cache and the total number of page faults (including both cache

Table 5.5: Prefetching contribution and accuracy when different Spark and Neo4j co-run with native applications.

Contribution	**Spark-LR**	**Spark-KM**	**Spark-TC**	**Neo4j**
Leap	23.4%	25.8%	42.2%	**67.0%**
Kernel	63.3%	68.0%	65.9%	41.1%
Canvas Two-tier	**79.2%**	**79.3%**	**75.3%**	45.0%
Accuracy	**Spark-LR**	**Spark-KM**	**Spark-TC**	**Neo4j**
Leap	16.8%	17.2%	35.9%	6.1%
Kernel	95.6%	96.4%	93.9%	80.4%
Canvas Two-tier	**94.3%**	**94.8%**	**94.9%**	**87.1%**

hits and demand swap-ins). Accuracy is defined as a ratio between the number of page faults hitting on the swap cache and the total number of prefetches. Clearly, contribution has a strong correlation with performance while accuracy measures the pattern recognition ability of a prefetcher. For example, for a conservative prefetcher that prefetches pages only if a pattern can be clearly identified, it can have a high accuracy (*i.e.*, prefetched pages are all useful) but a low contribution (*i.e.*, the number of prefetches is small).

Here we report prefetching contribution and accuracy for three prefetchers: Leap (on our isolated swap system), the kernel prefetcher (also on our isolated swap system), and Canvas's two-tier prefetcher. Among the three prefetchers, for all but Neo4j, Leap has the lowest accuracy and contribution because it is an aggressive prefetcher. Leap keeps prefetching pages even when it cannot detect any patterns, which greatly reduces the prefetching accuracy. Second, due to the limited swap cache, the useless pages prefetched can cause previously prefetched pages to be released before they are accessed, hurting contribution. The kernel prefetcher and Canvas have comparable accuracy because the kernel prefetcher is much more conservative than Leap. It stops prefetching when no clear pattern can be observed. However,



(a) Latency CDF.

Without horizontal scheduling	
Contribution:	52.1%
Accuracy:	46.9%
With horizontal scheduling	
Contribution:	**62.8%**
Accuracy:	**52.4%**

(b) Prefetching effectiveness.

Figure 5.13: Horizontal scheduling effectiveness for GraphX-CC: (a) prefetching latency reduced, and (b) prefetching contribution and accuracy improved.

Linux has lower contribution than our two-tier prefetcher since Canvas prefetches more useful pages using semantics.

5.5.4.3 RDMA Scheduling

We evaluate our two-dimensional RDMA scheduling. For the vertical dimension, we use the weighted min-max ratio (WMMR) $\frac{\min(x_i/w_i)}{\max(x_i/w_i)}$ [234] as our bandwidth fairness metric (the closer to 1, the better), where x_i is the bandwidth consumption of the application i, and w_i is its weight. We set the weight proportionally to the average bandwidth of each application when running individually. Our vertical scheduling achieves an overall of **0.88** WMMR.

The horizontal dimension (*i.e.*, priority scheduling with timeliness) is our focus here because interference between prefetching and demand swapping is a unique challenge we overcome in this work. We ran GraphX Connected Components (GraphX-CC) with the three native applications. Figure 5.13 compares the latency of sync vs. async swap-in requests with and without the horizontal scheduling of RDMA.

As shown, our scheduler does *not* incur overhead for the synchronous, demand requests but reduces the (90$^{\text{th}}$ percentile) latency of the asynchronous prefetching requests by ~5%. Note that these results were obtained with Canvas's two-tier prefetcher enabled, which already generates precise prefetching requests. With the Leap prefetcher, the (90th percentile) latency reduction can be as high as **9×**. To understand how the latency reduction improves prefetching effectiveness, we have also compared the prefetching contribution and accuracy with and without the horizontal scheduling, as shown in Figure 5.13(b). Due to the high timeliness requirement of prefetching requests, even 5% latency reduction can lead to noticeable improvements in prefetching—*e.g.*, the contribution/accuracy of GraphX-CC increases by **10.7%** and **5.5%** on top of the two-tier prefetcher—which ultimately translate to a **7-12%** overall improvement.

5.6 Related Work

In addition to the resource disaggregation and remote memory work discussed in §2.2 and §4.6, Canvas is also related to other areas of research, as outlined below.

Resource Isolation. Interference exists in a wide variety of settings [65, 149, 281] and resource isolation is crucial for delivering reliable performance for user workloads. There is a large body of work on isolation of various kinds of resources including compute time [28, 50, 141], processor caches [79, 120, 262], memory bandwidth [107, 146, 147, 154, 270], I/O bandwidth [91, 151, 159, 234, 244, 257, 271], network bandwidth [25, 83, 92, 113, 173, 203, 232], congestion control [60, 98], as well as CPU involved in network processing [122]. Techniques such as IX [29] and MTCP [112] isolate data-plane and application processing at the core granularity.

Prefetching. Prefetching has been extensively studied, in the design of hardware cache [93, 167, 250, 251, 286], compilers [70, 125, 131, 200, 211, 245], as well as operating systems [158, 255]. Detecting spatial patterns [160] is a common way to prefetch data. For example, various hardware techniques [109, 117, 231] have been developed to identify patterns (*i.e.*, sequential or stride) in addresses accessed. Leap [158] is a kernel prefetcher designed specifically for applications using remote memory. Swap interference can reduce the effectiveness of any existing prefetchers, let alone that none of them consider complex (semantic) patterns. Early work such as [39, 199] proposes application-level prefetching for efficient file operations on slow disks. Our prefetcher is, however, designed for a new setting where applications trigger page faults frequently and read pages from fast remote memory, with much tighter latency budgets.

RDMA Optimizations. There is a body of recent work on RDMA scheduling [210, 227] and scalability improvement [49, 118, 119, 248, 284]. These techniques focus more on scalability when RDMA NICs are shared among multiple clients.

5.7 Summary

We observed that swap resources must be isolated when multiple applications use remote memory simultaneously. As such, Canvas isolates swap cache, swap partition, and RDMA bandwidth to prevent applications from invading each other's resources. Now that resource accounting is done separately for applications, Canvas offers three optimizations that adapt kernel operations such as swap-entry allocation, prefetching, and RDMA scheduling to each application's resource usage, providing additional performance boosts.

Canvas concludes our exploration of memory harvesting techniques. Along with Midas and Hermit, Canvas forms a comprehensive solution for applications to efficiently and safely harvest memory, whether it is local or remote, shared or not. These systems provide a solid foundation for distributed memory management, opening opportunities for higher-level resource-harvesting programming frameworks (discussed in §7.1). However, their capabilities are limited to resources managed directly by the OS, such as CPU and memory. Given the growing popularity of AI and GPU-driven applications, where specialized runtimes bypass the OS to access GPUs, current resource harvesting systems fall short. In the next chapter, we overcome this limitation by applying our insights to harvest GPU resources for AI workloads.

CHAPTER 6

Concerto: Harvesting GPUs for Large Language Model Serving

The rise of large language models (LLMs) and recent AI breakthroughs have significantly reshaped datacenter workloads, increasing the demand for accelerators like GPUs alongside traditional resources such as CPU and memory. Given the high operational costs of GPUs, optimizing their utilization has become increasingly critical. Although managing GPU resources differs greatly from managing traditional resources, we argue that our earlier insights remain applicable and effective for harvesting idle GPU resources.

In this chapter, we revisit the first insight from Section 1.2, identifying offline inference tasks—such as document summarization and information extraction—as the elastic component of LLM serving. To efficiently harness idle GPU resources, we introduce Concerto, a preemptive GPU runtime designed for large language model serving that co-locates online and offline inference tasks. Concerto uniquely leverages available GPU resources left unused by online tasks to opportunistically batch offline inference, thereby maximizing GPU utilization. During online load bursts, it reactively preempts offline tasks, ensuring low latency for online serving. Through this approach, Concerto simultaneously achieves high throughput, low latency, and optimal GPU utilization for large language model serving.

6.1 Introduction

Large language models (LLMs) such as ChatGPT [189], LLaMA [247], and GPT-4 [190] have emerged as transformative tools across a wide spectrum of application domains. LLM-powered services, *e.g.*, interactive chatbot [51, 189, 242], programming assistant [84, 110, 215], and document summarization tools [170, 175], have already shown promises to businesses and end consumers [101] and are expected to create increasing impact in all aspects of human lives.

However, the superior abilities of LLMs come with high computational and memory demands to serve LLM inference requests. User requests to LLMs are served by running model inferences on GPUs, which can be significantly more expensive than typical web requests. To make matters worse, many LLM services such as chatbots are hosted *online*, requiring low response latency for user experience. To meet the strict latency service level objectives (SLOs), operators often need to overprovision GPUs to the LLM service, causing significant resource waste.

The tension between low tail latency and high resource utilization is further exaggerated by the bursty load patterns commonly exhibited in LLM workloads. LLM load varies not only over long timescales of hours, but also over timescales as short as a few seconds. For example, a recent study [263] reported that the load to ChatGPT can increase by 3× within one minute. As a result, the service provider must provision GPUs to the peak user load to prevent vast latency SLO violations under load bursts.

A parallel trend is that LLM-based applications are evolving into compound AI systems [275], which involve retrieval-augmented generation (RAG) [201, 280], tool usage [130, 191], SQL-based data analytics [148], *etc.* Other than simply using LLMs to reply to online user requests, compound AI systems often use LLMs for *offline batch* inference, which has loose latency requirements but desires high generation throughput in a best-effort manner[1].

[1]For brevity, herein we refer to latency-critical requests as *online*, and best-effort requests that are latency insensitive as *offline*.

While many recent LLM serving systems are aimed at optimizing the inference efficiency [10, 99, 128, 198, 272, 285], they operate under the assumption that the workload follows a constant request rate and latency requirement and hence may fall short when dealing with real-world workloads, which are bursty and heterogeneous (*i.e.*, with both online and offline requests and drastically different SLOs). As a result, to deploy these systems, today's datacenter operators must use separate clusters for serving online and offline requests, and overprovision GPUs for online serving to meet latency SLOs [209].

The conventional wisdom to reduce resource waste is to dynamically repartition the cluster in response to the load variation of online requests [46, 142]. However, it requires *a priori* knowledge of the online load variation patterns, which are often hard to predict. Allocation based on inaccurate load estimation can either be too conservative (still resulting in overprovisioning) or too relaxed (leading to SLO violations). In addition, re-allocating GPUs across different jobs is a complicated task that involves tearing down an existing process, cleaning up GPU resources, launching a new process with adjusted parameters, and setting up GPUs for serving. These operations are often time-consuming and take seconds to minutes to finish, preventing the system from rapidly reacting to load bursts and preserving tight latency SLOs.

Insights. The key question we ask in this chapter is: instead of partitioning the cluster and serving online and offline requests separately, can we *co-serve them with the same set of GPUs*? In other words, our goal is no longer to adapt GPU allocation to online load variation, but instead to adapt offline serving throughput to the available GPU resources. In doing so, we can maximize the GPU utilization while shifting the need for GPU reallocation to offline serving, which can tolerate relatively high response latency.

To achieve this goal, we developed *Concerto*, a unified LLM serving system that serves online and offline requests simultaneously and dynamically coordinates GPU resources between them. Concerto frees the service provider from the burden of manually allocating and adjusting

GPUs for online serving. Instead, it achieves high GPU utilization by opportunistically scheduling offline requests whenever GPU compute resources and memory are available. To avoid resource contention that may break the latency SLO of online serving, Concerto proactively *preempts* offline requests and rapidly reclaims resources in response to online load bursts. Concerto recovers preempted requests after online requests are handled.

Challenges. The idea of co-locating latency-critical jobs with best-effort jobs has been widely studied in traditional cloud workload scheduling [46, 76, 194]. However, to realize its benefits in LLM serving, Concerto needs to overcome several unique challenges.

First, how can the system quickly free up resources taken by offline serving in response to online load bursts? Since LLM inference is iterative, a natural idea is to preempt offline requests after a generation iteration. However, the online request rate is highly unpredictable and they come with a tight latency requirement (usually in milliseconds). Serving a large batch of offline requests, even for a single iteration, may block incoming online requests for seconds.

To solve this problem, Concerto preempts running offline requests at the (fine) granularity of model layers. We found that the layer granularity strikes a good balance between responsiveness and execution efficiency. On the one hand, for different LLMs their layer sizes are generally small and do not vary much, allowing Concerto to discard partial results and reclaim occupied GPU resources much faster than finishing an entire iteration. On the other hand, compared to choosing a finer granularity such as CUDA kernels [97], instrumenting a model on a per-layer basis is lightweight, incurring only negligible overhead (see §6.4.3).

Second, how can the system minimize the recomputation cost? While the system can react to online load bursts with preemption, it discards the intermediate states (*i.e.*, KV cache) of the preempted offline requests. Consequently, it requires expensive recomputation to recover the lost KV cache once the preempted requests are resumed. A natural idea is to swap out the KV cache to host memory. Unfortunately, the size of the KV cache can grow unbounded, and swapping a large amount of data may, again, block incoming online requests.

Our insight is that LLM inference is stateless, and the KV cache remains unchanged once generated. Leveraging this property, Concerto incrementally checkpoints the KV cache of offline requests to the host memory per generation iteration. Since one token is generated per request, the total amount of data to be written to the host memory is small and bounded. Concerto further makes checkpointing asynchronous by overlapping it with the computation (see §6.4.4).

Finally, how can the system preserve the latency SLO for running online requests? While batching offline requests improves GPU utilization, the batch size must be adjusted dynamically to ensure that online requests in the same batch can meet the latency SLO. This is, however, challenging because the inference latency is the result of many factors including the batch size, the total number of tokens, the phase of each request (*i.e.*, prefill or decode), *etc.* To overcome this challenge, Concerto uses a profiler and a SLO-aware scheduler. The profiler runs offline and collects the execution time of different input batch sizes and input lengths for requests in different stages. The scheduler adaptively changes the number of offline requests and tokens based upon the profiling results and the number of online tokens (see §6.4.5).

Results. We evaluated Concerto with two real-world datasets and synthetic workloads. Our results demonstrate that Concerto achieves comparable latency and throughput to vLLM [128] (a state-of-the-art LLM serving system), but can trade off online inference throughput for offline inference throughput in a linear fashion when the online request rate is lower than the peak load. With GPUs harvested for offline requests, Concerto achieves 2.35× higher throughput than vLLM with the same 99th percentile latency, and outperforms existing co-serving systems by 84× in terms of serving latency.

6.2 Background

6.2.1 Large Language Model Inference

Today's LLMs typically generate outputs following the *autoregressive* process—they take a sequence of tokens as input, and repeatedly predict the next token given the input sequence and all the previously generated tokens. This process involves two distinct phases: `prefill` and `decode`. An LLM inference starts with the `prefill` phase that deals with a new input sequence and generates the first output token. Thanks to the modern model architecture such as Transformer [252], an LLM can process all input tokens in parallel, making the `prefill` phase compute-bound. After the `prefill` phase, the LLM enters the `decode` phase that sequentially generates subsequent tokens. Because each `decode` iteration generates only one token, it is less compute-intensive and typically bounded by the GPU memory bandwidth due to the need to load model weights.

Each token during the inference has its own intermediate state, represented by a series of key vectors and value vectors. As token states remain unchanged throughout the inference process, the inference engine [185, 186, 193] caches them in GPU memory (also known as KV cache) to avoid recomputation in each `decode` step. However, due to the large size of KV vectors and the excessive number of tokens in a batch, the KV cache can consume a significant amount of GPU memory.

6.2.2 Characterizing LLM Serving

The rise of compound AI systems and LLM agents dictates that LLMs be used for various demands and in different forms. Generally, LLM serving can be categorized into two types: online serving which generates responses to user inputs in real-time, and offline serving which processes user inputs and generates outputs in a batch. Typical scenarios for offline serving including document summarization [115], LLM benchmarking and evaluation [143], data wrangling [174], and LLM-enhanced data analytics [148]. However, online and offline serving expose drastically different characteristics, as elaborated below.

Online serving is mostly suitable for latency-critical requests. Unlike traditional cloud services that take constant processing time, a request to an LLM may lead to a sequence of tokens generated in multiple steps. Therefore, the serving latency is measured on a per-token basis. Moreover, because LLM inference consists of two distinct phases, an LLM's serving latency is defined by two metrics: *time to first token (TTFT)*, which is the duration of the `prefill` phase, and *time per output token (TPOT)*, which is the execution time of each `decode` phase. It is important for an online LLM serving system to optimize both TTFT and TPOT to reduce the end-to-end request latency, and the service often requires tight SLOs for TTFT and TPOT to provide a smooth user experience. For example, an online chatbot may set its 99th percentile TTFT SLO to 1.25 seconds and 99th percentile TPOT SLO to 100 milliseconds to exceed typical human reading speed [172, 205].

In contrast, offline serving is a better fit for best-effort requests that are insensitive to response latency. Users usually submit a batch of offline requests as a batch processing job, while the serving system processes offline requests in a best-effort manner for maximized hardware efficiency. Because offline requests often come from a large corpus or request pool, the first-order metric for offline serving is *throughput*, which measures how many tokens are generated per second.

The different service-level performance objectives between online and offline serving poses challenges to LLM serving systems. To overcome the tension between low response latency (TTFT and TPOT) and high generation throughput, today's LLM serving systems usually adopt different configurations and designs for different serving scenarios, typically requiring the service provider to set up separate clusters for online and offline serving. Next, we will discuss how existing systems optimize LLM serving and why they fall short in achieving high GPU utilization.

6.2.3 Existing LLM Serving Systems

A large body of system optimizations have been proposed to serve LLMs with low latency and high throughput since the launch of ChatGPT.

Throughput-Oriented Optimizations. Orca [272] is one of the pioneer LLM serving systems that employ *continuous batching* to improve inference throughput. It adds incoming requests directly into existing running batches, achieving higher throughput with a larger batch size. However, because new requests are in their `prefill` phase while running requests are in their `decode` phase, Orca sacrifices TPOT due to the larger batch size and additional `prefill` computation. In this same direction, vLLM [128] further enlarges the batch size by reducing the KV cache fragmentation and hence the GPU memory consumption.

Latency-Oriented Optimizations. Recently, more systems have been proposed to optimize LLM inference latency. Among them, Sarathi-Serve [10] and Deepspeed-FastGen [99] piggyback on the continuous batching strategy, but break the *prefill* phase of incoming requests into small, fixed-size chunks (chunked-prefill), and only add one chunk into the running batch in each step. This limits the cost of additional prefill-phase computation in each generation step, thereby improving TPOT. However, as a new request is broken into many small chunks and processed in multiple steps, this line of work often leads to sacrificed TTFT.

Another line of work moves away from continuous batching scheduling, as exemplified by DistServe [285] and Splitwise [198]. Instead, they duplicate the model and disaggregate the `prefill` and the `decode` computation onto different GPUs. Specifically, new requests are sent to a dedicated cluster for the *prefill* computation, and the generated token and KV cache are populated to another dedicated cluster that performs the *decode* computation. Llumnix [238] is another recent work that migrates KV caches between servers to reduce load imbalance. These systems achieve low TTFT and TPOT at the cost of more GPUs. Due to the additional data transfer of KV caches between servers, they also lead to suboptimal throughput.

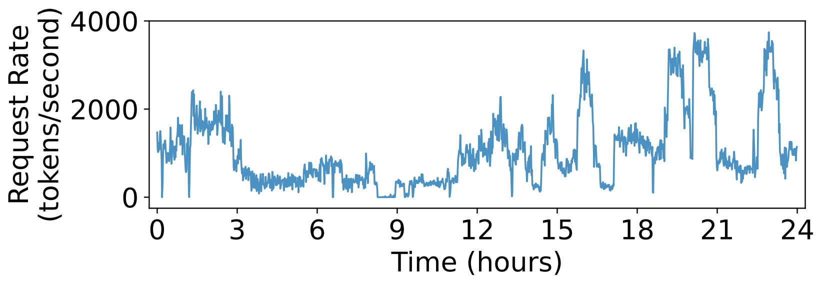
Limitations of Existing Techniques. Existing LLM serving systems suffer from two major limitations. First, they trade off between latency and throughput, and hence are only applicable for either online or offline serving. Second, they can neither determine how many GPUs should be provisioned, nor adaptively change the number of GPUs during serving. In fact, because they target only one type of incoming requests, they often assume a constant request arrival rate and reserve enough GPUs in advance.

Unfortunately, these problems are more pronounced when serving real-world inference workloads, which can contain both online and offline requests with high load variation. Such a challenge makes it hard to provision the right amount of GPU resources *a priori*.

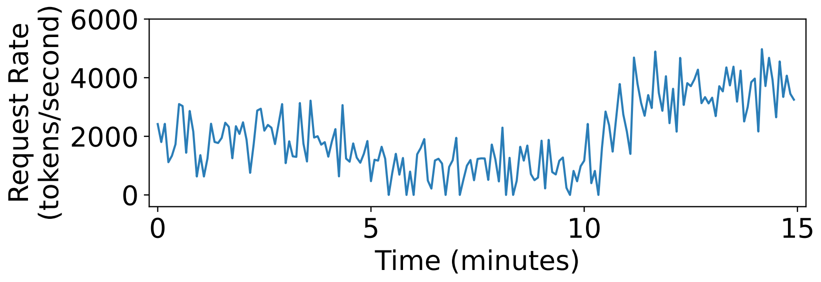
6.3 Motivation

In this section, we first demonstrate how real-world LLM load can vary over time and why existing serving systems fall short of achieving high GPU utilization. We then use an experiment to quantitatively demonstrate why simply co-locating online serving with offline serving cannot solve the problem.

Online Load Burstiness. Real-world LLM workloads often expose diurnal patterns and high load variability, as backed by a recent study [263] which collects user traffic to ChatGPT for two months within a campus. Figure 6.1a shows the load variation within a day. Despite the average load being as low as 1050 tokens per second, there is a clear contrast between peak hours and non-peak hours. The load can achieve more than 3743 tokens per second in the afternoon, while in the morning the service only experiences little traffic. In addition, there are unpredictable load bursts within an hour, during which the request rate can ramp up multiple times in a short period. Figure 6.1b further provides a closer examination of



(a) load variation over 24 hours.



(b) Load variation over 15 minutes.

Figure 6.1: User traffic to ChatGPT within a campus exposes high load variability at various time scales.

one such burst in a 15-minute window. As shown, the load still fluctuates drastically at the minute timescale, and the request rate increases by 3× in the tenth minute.

Due to such high load variability, service operators must reserve resources for peak demand to avoid violating latency SLOs. However, since the peak inference load can be multiple times higher than the average load, overprovisioning can lead to significant resource waste.

Naïve colocation hurts online serving latency. A simple strawman approach is to extend today's online serving systems to allow users to submit requests with priorities. Users may assign online requests a high priority and offline requests a low priority. The scheduler should incorporate the priority information and schedule online requests first to preserve low latency. Since none of the existing systems support co-serving online and offline requests, we implemented a priority-based scheduler atop vLLM [128] (details in §6.6.1) and used it

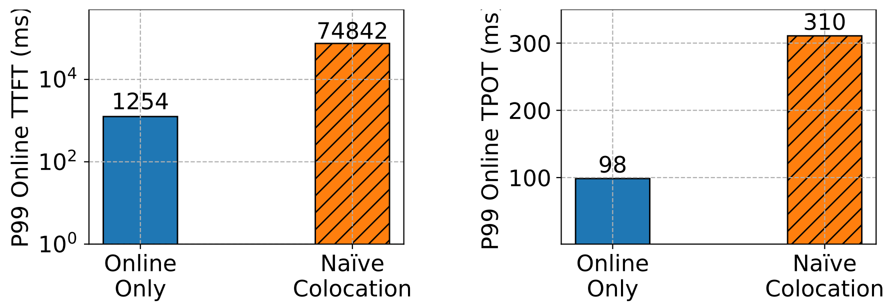
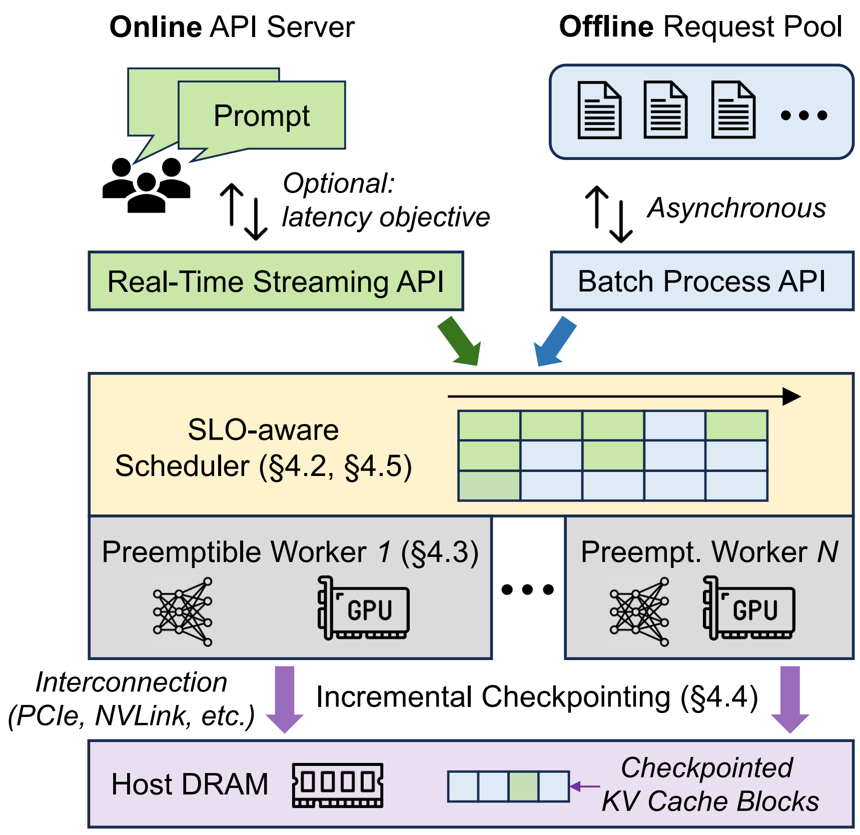


Figure 6.2: 99th-percentile TTFT and TPOT of online requests when co-located with offline requests using a naïve priority-based scheduler. Note that the *y-axis* of TTFT is displayed on a log scale. Due to severe interference, Naïve colocation ramps up the 99th percentile latencies for online requests by one to two orders of magnitudes.

to serve Llama-2 7B on one Nvidia A100 GPU. We replayed the trace in Figure 6.1 and collected the 99th-percentile TTFT and TPOT for online requests. These results are reported in Figure 6.2.

While co-serving offline and online requests has indeed improved GPU utilization, it could significantly impact online serving latency—P99 TTFT increases by 59.6× and P99 TPOT increases by 3.16×. There are two reasons for the latency increase. First, once offline requests are scheduled, they are batched together with online requests and cannot be preempted selectively. Therefore, incoming online requests must wait until they are served, leading to significantly increased queueing delays. Second, the scheduler tends to pack enough offline requests to make full use of GPU memory, which can lead to large batch sizes that take longer to finish, and hence a further increased inference latency.

These problems call for a new system that can harvest available GPU resources for offline serving *on-the-fly* and dynamically adjust the amount of resources allocated to different types of requests to preserve online serving SLOs.



Online API Server
Offline Request Pool
Prompt
Optional: latency objective
Asynchronous
Real-Time Streaming API
Batch Process API
SLO-aware Scheduler (§4.2, §4.5)
Preemptible Worker 1 (§4.3)
Preempt. Worker N
GPU
GPU
Interconnection (PCIe, NVLink, etc.)
Incremental Checkpointing (§4.4)
Host DRAM
Checkpointed KV Cache Blocks

Figure 6.3: Overall achitecture of Concerto. Concerto efficiently co-serves online and offline requests with three major components: an SLO-aware scheduler, a set of preemptive workers, and an incremental checkpointing mechanism.

6.4 Design

6.4.1 Concerto Overview

Concerto is an LLM serving system designed to co-serve online and offline requests. Figure 6.3 shows its overall architecture. At its frontend, Concerto provides similar APIs to other LLM serving systems. For online requests, it uses a real-time streaming API that returns outputs once each token is generated. For offline requests, it adopts an interface similar to OpenAI's Batch API [192], which takes a batch of requests from a pool and returns responses asynchronously.

Concerto relies on its backend to schedule and execute inference jobs, which consists of three major components. The scheduler runs as a daemon thread and continuously fetches and schedules offline requests. Upon the arrival of an online request, the scheduler reactively preempts offline requests and immediately schedules the incoming online request (§6.4.2). The scheduler leverages a set of workers to host the LLM and serve the scheduled batch. A model can span multiple GPUs, where each GPU is managed by one worker. Concerto supports both tensor and pipeline parallelism for high efficiency and flexibility (§6.4.3). To quickly recover preempted requests with minimized recomputation costs, Concerto leverages host memory and incrementally checkpoints KV caches during its execution (§6.4.4). Finally, because Concerto can schedule both online and offline requests in the same batch, it adopts an SLO-aware policy to adaptively adjust the batch size to preserve online TTFT and TPOT latency objectives while maximizing the offline serving throughput as well as hardware efficiency (§6.4.5).

6.4.2 Unified Preemptive Scheduler

Concerto's scheduler is the key component that serves both online and offline requests in a unified fashion. As with previous LLM serving schedulers [128, 272], Concerto adopts continuous batching in its scheduler. To prevent long prefills from interfering decode iterations, Concerto also adopts chunked prefill [10, 99] that partitions and computes long prefills into small chunks over multiple iterations and limits the batch size per iteration.

The overall scheduling logic is shown in Algorithm 1. It maintains two separate queues for online and offline requests, and continuously monitors the online queue for incoming requests. In each scheduling step, it first calculates a budget batch size given the latency objectives for TTFT and TPOT (Line 8). The budget limits both the number of tokens and the number of requests in a batch. The detailed policies are elaborated in §6.4.5. The scheduler then checks and schedules incoming online requests within the budget allowance (Lines 9-13). To prevent previously scheduled offline requests from blocking incoming online requests, the

Algorithm 1: The unified scheduling logic for online/offline requests. Concerto prioritizes online requests, and only schedules offline requests when remaining GPU resources permit.

```
Input: time to first token (TTFT) objective $t_{TTFT}$, time per output token (TPOT)
       objective $t_{TPOT}$.
1  Global $Q_{on}$ (online request queue), $Q_{off}$ (offline request queue), $Q_{out}$ (output queue)
2  Global $t_{sched}$ (time when last batch gets scheduled)
3  Function UnifiedSchedule($Q_{on}$):
4      Initialize token budget $\tau \leftarrow Inf$
5      Initialize current scheduled batch $B \leftarrow \emptyset$
6      Initialize new online requests has_new_online $\leftarrow$ False
7      while True do
8          $\tau \leftarrow$ calc_budget($t_{TTFT}$, $t_{TPOT}$) - $B$.num_tokens()
9          if !$Q_{on}$.is_empty() then
10             $\tau_{off} \leftarrow B$.num_offline_tokens()
11             $B_{on}, \tau \leftarrow$ SchedChunkedPrefill ($Q_{on}$, $\tau + \tau_{off}$)
12             $B \leftarrow B \bigcup B_{on}$
13             has_new_online $\leftarrow$ True
14         $B$, $\tau \leftarrow$ PreemptOverBudgetOffline ($B$, $\tau$)
15         if $B$.has_online_requests() then
16             $B_{off}, \tau \leftarrow$ SchedChunkedPrefill ($Q_{off}$, $\tau$)
17         else
18             $B_{off}, \tau' \leftarrow$ SchedChunkedPrefill ($Q_{off}$, $Inf$)
19         $B \leftarrow B \bigcup B_{off}$
20         $t_{sched} \leftarrow$ time.now()
21         $B, B_{finished} \leftarrow$ exec_batch($B$)
22         $Q_{out}$.append($B_{finished}$)
```

Algorithm 2: Concerto iteratively preempts offline requests when GPU is under compute pressure (*over_ budget()*) or memory pressure.

```
1 Function PreemptOverBudgetOffline ($B$, $\tau$):
2     for $R$ in $B$.offline_reqs() do
3         if not $\tau$.over_budget() and gpu_memory_sufficient() then
4             break
5         PreemptScheduling ($R$)
6         $B \leftarrow B \setminus \{R\}$
7         $\tau \leftarrow \tau - R$.num_tokens()
8     return $B$, $\tau$
```

scheduler excludes offline tokens from the budget first (Line 10-11), and then reactively preempts offline requests if the budget is over-saturated (Line 14). This preemption continues until all scheduled requests can fit into the budget (Lines 2-8). After accommodating all online requests, the scheduler opportunistically schedules offline requests using the remaining budget (Line 15-16). The detailed scheduling policies (*i.e.*, ScheduleChunkedPrefill) to maintain online latency SLOs and manage GPU memory usage will be discussed shortly in §6.4.5.

Offline Batching Mode. Due to the diurnal load patterns, a model may not receive any new online requests periodically during non-peak hours. During such periods, the scheduler switches to the *offline batching mode* for maximizing offline serving throughput (Line 17-18 in Algorithm 1). Because offline requests come with only loose or no latency requirements, the scheduler ignores the budget limit and sets the largest batch size that can saturate GPU compute or memory capacity.

Algorithm 3: Preemptive scheduling logic: the arrival of an online request triggers a callback function and preempts running workers if necessary to meet the TTFT objective.

```
Input: Incoming online request $o$.
1  Global $Q_{on}$ (online request queue)
2  Global $t_{sched}$ (time when last batch gets scheduled)
3  Function OnRecvOnlineRequest($o$):
4      Initialize current time $t_{curr} \leftarrow$ time.now()
5      $Q_{on}$.append($o$)
6      $B \leftarrow$ Worker.get_curr_batch()
7      $t_{exec} \leftarrow$ Profiler.estimate_exec_time($Q_{on} \bigcup B$)
8      $t_{est} \leftarrow$ Profiler.estimate_exec_time($B$)
9      $t_{remain} \leftarrow t_{est} - (t_{curr} - t_{sched})$
10     if $t_{remain} + t_{exec} > t_{TTFT}$ then
11         break
12     $B_{victim} \leftarrow \emptyset$
13     for $R$ in $B$.offline_reqs() do
14         $B \leftarrow B \setminus \{R\}$
15         $B_{victim} \leftarrow B_{victim} \cup \{R\}$
16         $t_{exec} \leftarrow$ Profiler.estimate_exec_time($Q_{on} \bigcup B$)
17         $t_{est} \leftarrow$ Profiler.estimate_exec_time($B$)
18         $t_{remain} \leftarrow t_{est} - (t_{curr} - t_{sched})$
19         if $t_{remain} + t_{exec} \leq t_{TTFT}$ then
20             break
21     PreemptRunning ($B_{victim}$)
```

Preemption. Concerto can preempt offline requests at two potential points: during scheduling or model execution. First, in each scheduling step, the scheduler needs to preempt scheduled offline requests to make room for incoming online requests or free up GPU memory under memory pressure. Similar to prior systems such as vLLM [128], preempting a request during scheduling can be done by either discarding-and-recomputing or swapping to host memory, and it is implemented in the PreemptScheduling function at Line 5 in Algorithm 2.

However, in the worst case where incoming online requests are about to exceed their TTFT objectives, Concerto must deal with the urgent case and preempt offline requests even if they are in a running batch to schedule new requests in a timely fashion. To this end, Concerto scheduler invokes an asynchronous handler upon the arrival of new online requests, as shown in Algorithm 3. The handler first pushes the incoming online request into the online request queue, and then leverages the profiler (discussed shortly in §6.4.5) to estimate the queuing delay and the execution time (Lines 7-9). If the estimated serving time exceeds the TTFT objective, the scheduler signals the worker to preempt offline requests in the current running batch until it can meet the TTFT objective (Lines 10-21).

6.4.3 Preemptible Worker

To host the model on GPUs and execute inference requests, Concerto leverages a set of workers to manage GPUs. Concerto supports both tensor and pipeline parallelism in the Megatron-LM fashion [233]. Additionally, Concerto's workers support preempting offline requests in a running batch (*i.e.*, PreemptRunning in Algorithm 3 Line 21). But unlike previous work [97] that instruments and preempts running GPU kernels, Concerto worker preempts LLM inference at the granularity of *model layers*, which strikes a balance between responsiveness and runtime costs.

A unique challenge here is that we must synchronize all workers before preemption, because workers in the same tensor parallel group perform collective communication operations—simply preempting one worker would hang the other involved workers and hence the entire program. Therefore, Concerto must synchronize all workers in the same tensor parallel group before preemption. However, it is important to preempt at the right temporal granularity—if it is too coarse-grained, the system may not be able to react to load bursts timely; but if it is too fine-grained (*e.g.*, per GPU kernel as in previous work [97]), synchronization can incur high runtime overheads and harm overall efficiency.

To overcome this challenge, Concerto chooses to preempt at the granularity of *model layers* for two reasons. First, while their architecture and size may vary drastically, LLMs all consist of many layers, and the execution time for each layer is much shorter than the end-to-end inference latency; as such, preemption can be made responsive. Second, one LLM layer still consists of many GPU kernels and collective operations, and hence, preemption (including additional cross-worker synchronization) only incurs negligible runtime overheads.

Concerto uses a *master worker* for the cross-worker synchronization. When the scheduler decides to preempt the current batch, it signals the master worker and specifies a victim set of offline requests to be preempted. The master worker then synchronizes all workers with a barrier after finishing its current model layer and broadcasts the preemption signal as well as the victim set. Upon receiving the signal, all workers discard all requests in the victim set before continuing the execution of the next layer.

To further amortize the cost when the layer execution time is too short, a Concerto worker can also adjust the preemption granularity by batching multiple layers before the synchronization barrier.

6.4.4 Incremental Checkpointing

While preemption helps Concerto achieve low scheduling delay for online requests, it comes with a cost to recompute the discarded KV cache for victim offline requests that are preempted during scheduling or execution. Therefore, we must minimize the recomputation cost to fully unleash the GPU compute power for meaningful work.

A strawman approach applied by existing serving systems such as vLLM [128] is to swap out KV caches of victim requests to host memory, as shown in Figure 6.4(b). However,

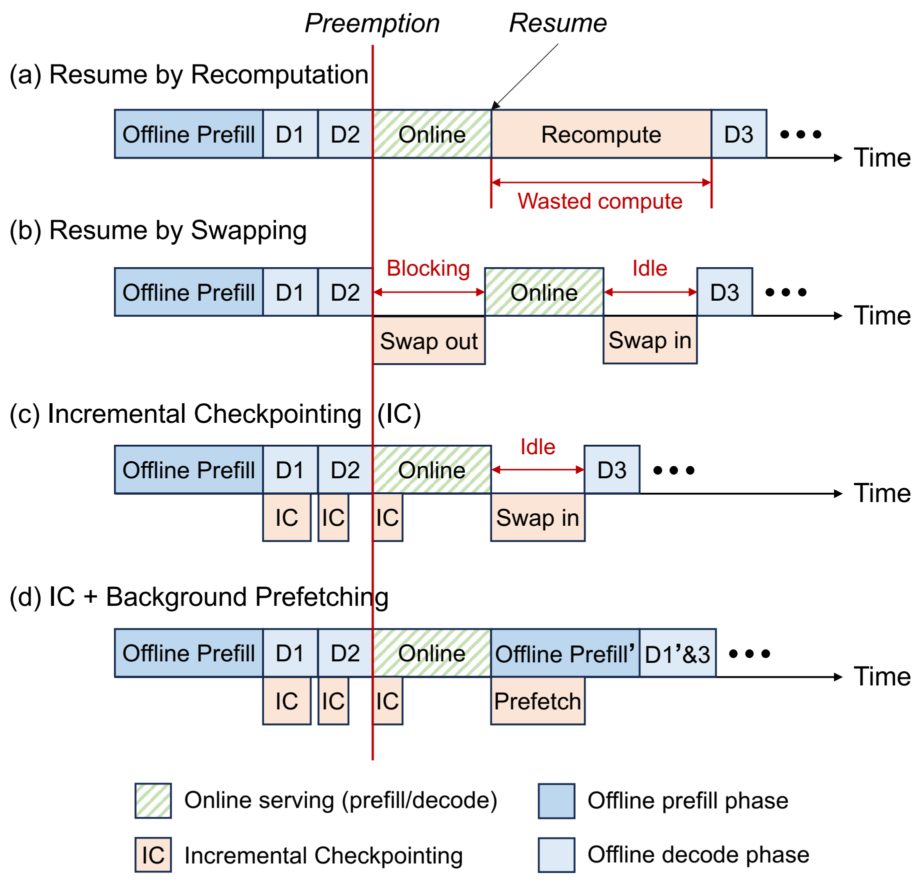


Figure 6.4: Comparison between different preemption and resume strategies. (a) Resume by recomputation achieves low preemption delay at the cost of additional computation. (b) Resume by swapping reduces the recomputation cost but swapping out can block the schedule of incoming online requests. (c) Incremental checkpointing (IC) minimizes both preemption delay and resume cost. (d) IC + background swap-in overlaps swap-in with prefill computation of the next batch and achieves consistently high GPU utilization.

swapping out by itself still takes time and blocks the scheduling of incoming online requests. To make matters worse, the amount of data to be swapped out increases proportionally to the number of tokens in offline requests, but the interconnection bandwidth between GPUs and host memory is limited. For example, Nvidia A100 connects with the host DRAM with PCIe 4.0x16, which offers only 32 GBps bandwidth. Swapping out all KV caches for offline requests can easily take dozens or even thousands of milliseconds, which still significantly blocks incoming online requests. Besides, to resume serving victim requests, the GPU must swap in evicted KV cache blocks before continuing the computation, which leaves GPU compute units idle and hurts overall serving throughput.

Incremental Checkpointing. To overcome these challenges, we propose a novel asynchronous checkpointing and resuming mechanism. Firstly, instead of swapping KV caches out at the last minute when preemption happens, Concerto worker adopts a checkpointing mechanism that runs asynchronously and incrementally in the background. As shown in Figure 6.4(c), Concerto asynchronously checkpoints KV caches of offline requests after each generation iteration. Because modern LLMs follow the auto-regressive nature and generate tokens iteratively, workers can also *incrementally* checkpoint KV caches per iteration. As a result, the PCIe traffic is amortized over multiple iterations. Besides, because checkpointing has no data dependency with follow-up computation, it can be done asynchronously in the background and overlapped with computation. As a result, it incurs only negligible runtime overhead.

Concerto manages GPU memory similar to vLLM [128] by reserving GPU memory ahead of time and virtually mapping KV cache blocks to physical GPU memory. But unlike swapping which frees KV caches right after they are swapped out, checkpointing keeps KV caches in GPU memory until incoming online requests are scheduled. Discarding KV caches in Concerto is as fast and lightweight as freeing victim KV cache blocks and remapping new ones virtually, which finishes at microseconds timescale.

Background Prefetching. Secondly, because offline requests do not have strict latency constraints, they offer Concerto workers a unique opportunity to re-order requests and overlap swap-in with computation. As shown in Figure 6.4(d), instead of waiting for victim requests to be swapped in, Concerto worker launches a new offline batch and runs for its prefill phase, and meanwhile prefetches KV cache blocks in the background. Afterward, the worker will merge the new batch with prefetched requests and run their decode phase together. For long sequences that need to swap in a large amount of KV blocks to resume, Concerto will also split the swap-in phase over multiple steps to avoid long swap-in delays. In doing so, Concerto keeps device-host I/O for offline requests entirely in the background and eliminates idle GPU cycles.

Adaptive Checkpointing Policy. However, blindly applying the checkpointing technique will excessively use host memory and interconnection bandwidth, potentially causing resource contention when online requests also need swapping. Furthermore, checkpointing is only necessary under GPU resource pressure when offline requests are likely to be preempted. In other cases, it can be avoided to reduce runtime overhead. Inspired by the asynchronous swap system design in the OS kernel [207] and the conventional wisdom of random early detection [72], Concerto adaptively controls the checkpointing rate based on GPU memory pressure to limit resources used by checkpointing. Specifically, Concerto starts checkpointing when available GPU memory is running low (by default the threshold is set to 50%). but it will only checkpoint a small number of offline requests first and gradually increase the number of offline requests to checkpoint after observing constantly increasing memory usage. In most cases when online load bursts are not intensive, preempting checkpointed offline requests will make enough room to schedule incoming online requests and buy some additional time to checkpoint the remaining offline requests.

Finally, incremental checkpointing can also help online serving to accommodate swapping/preemption caused by continuous batching. With the continuous batching policy, the scheduler will eagerly add new requests to the batch to enlarge the batch size when GPU memory permits, but this may consume all GPU memory when requests in the batch generate long output sequences and keep allocating memory for KV caches. In such scenarios, the scheduler will have to swap or preempt online requests to make room. Concerto will also incrementally checkpoint online requests under GPU memory pressure, thereby eliminating the cost of swapping out or recomputation for online requests as well.

6.4.5 SLO-aware Scheduling

The last challenge that Concerto must address is to determine the right *batch size* (*i.e.*, the number of offline requests and tokens) to compute and the right *set of KV cache blocks* to checkpoint/prefetch for each inference iteration. (1) As for the batch size, if the scheduler batches too few offline requests, a substantial portion of GPU compute resources and memory bandwidth would be underutilized and lead to suboptimal overall serving throughput; on the flip side, if the scheduler batches too many offline requests, they will saturate the GPU and increase the inference computation time and hence online serving latencies, resulting in SLO violations. To make matters more complicated, the model execution time depends on not only the batch size but also the context lengths of each request due to the non-linear compute complexity of the attention kernel ($O(n^2)$ for prefill phase and $O(n)$ for decode phase where n is the context length)[2]. Therefore, the scheduler must also decide the number of offline tokens that can be batched and processed together with online tokens while meeting the SLO. (2) As for the KV cache blocks to checkpoint/prefetch, swapping too many blocks, even if in the background, will exhaust the PCIe bandwidth and GPU streaming multiprocessors (SMs) resources and block the computation kernel, while checkpointing too few blocks will make it unable to keep up with the GPU memory consumption rate, leading to memory exhaustion and triggering swap that blocks incoming online requests.

To tackle this challenge, Concerto adopts an SLO-aware scheduler to collect the model execution profiles with an *offline profiler* and dynamically adjust the batch size and the degree of background swapping leveraging the collected information and SLOs specified by the users. To flexibly batch varying numbers of offline tokens into each batch, Concerto leverages chunked prefill [10] to break offline sequences if necessary.

[2]Chunked prefill alleviates this effect by splitting long sequences into smaller chunks, but its performance is still prone to prefix lengths due to repeated KV cache access [10].

Profiler. Concerto's performance and concrete scheduling policy are highly dependent on the model itself and the hardware configurations, including specific GPU models, the parallelization strategy, PCIe bandwidth, *etc.*. To quantify the performance impact, Concerto will first run its profiler in the offline phase to profile the model prefill and decode computation time with different numbers of tokens, as well as the swap latency with respect to the number of KV cache blocks. The profiled results will be saved locally and automatically loaded when launching a Concerto server.

SLO-aware Policy. Concerto then leverages the profiled information to schedule offline requests. For each scheduled online batch, it queries the profiler with the latency SLO (TPOT for batches containing decode phase requests, TTFT otherwise) to get the maximum number of tokens that can be processed. It then schedules just enough offline tokens to fulfill the batch. Similarly, it uses the SLO to decide the maximum number of KV cache blocks that can be swapped in the background, and defers the extra blocks to the next round. In most cases where swapping a block is faster than computation, or offline tokens only take a small portion of the batch size, this policy is memory-safe and can always checkpoint KV cache blocks faster than the GPU memory consumption. Under the extreme case where a huge amount of KV cache blocks need to be checkpointed, Concerto will prioritize online latency SLO attainment and fall back to discard excessive KV cache blocks and recompute them later.

6.5 Implementation

We have implemented Concerto atop vLLM 0.4.2 [128] with 4165 lines of code. Concerto leverages Ray [171] to distribute the model to multiple GPUs, but it additionally supports a Python `multiprocessing` backend that communicates via shared memory to eliminate Ray's high inter-process communication (IPC) cost when running with multiple GPUs on a single node.

To support layer-wise preemption, Concerto instruments the model to add a preemption *safepoint* between layers, inspired by the design of managed language runtimes [5]. The safepoint contains a small piece of code that synchronizes all workers and then checks the `preempted` flag, which is a variable shared by the scheduler and all workers. If the worker detects `preempted` is set, it will abort the following layers and return directly.

Because Concerto now co-serves online requests and offline requests with a unified runtime, it does not need to reallocate GPU memory between two types of requests anymore. Instead, it leverages vLLM's paged KV cache management and reallocates only virtual KV cache blocks. To support asynchronous checkpointing, Concerto keeps track of the mapping between each GPU KV block and its CPU KV block that contains the corresponding checkpoint. Such mapping is recorded in an extension field of the virtual page table and can be queried and updated by Concerto scheduler.

Concerto also comes with a built-in load generator that can generate precisely timed request patterns following the gamma distribution. The load generator can be configured with various parameters including the request rate, burstiness (*i.e.*, skewness of the gamma distribution), and request lengths.

6.6 Evaluation

Our evaluation aims to answer the following questions:

1. Can Concerto judiciously coordinate GPU resources between online and offline serving to optimize overall performance? (§6.6.2)
2. Can Concerto quickly and reactively harvest available idle GPU resources to improve offline serving throughput? (§6.6.3.1)
3. Can Concerto quickly and reactively react to online load bursts and maintain low latency? (§6.6.3.1 and §6.6.3.2)

6.6.1 Setup

Environment. We conducted experiments on one server that equips a 48-core CPU, 340GB GB memory, and four NVIDIA A100-40G GPUs connected with 600 GBps fully-meshed NVLink. The server ran Ubuntu 22.04 and CUDA 12.1. To reduce latency jitter, we disabled dynamic voltage and frequency scaling (DVFS) of GPUs [6, 239] and Python garbage collector [7].

Baselines. We compared Concerto with vLLM [128], a state-of-the-art LLM serving system. We also enabled chunked-prefill in vLLM to ensure a fair comparison. Because the original vLLM cannot co-serve online requests and offline requests, we evaluated it by only feeding online requests (referred to as `Online-Only`), which provides the optimal online serving latency but zero offline serving throughput. To enhance its performance, we also extended vLLM's online serving frontend with a batch process API, so that it can also take batched offline requests while serving online requests. We additionally implemented a priority scheduler that prioritizes online requests over offline ones, and we refer to this enhanced baseline as `vLLM++`.

Models. We chose the Llama [247] model family, one of the representative open-source LLM series. Specifically, we tested the Llama-2 7B model on a single A100 GPU. All experiments use FP16 precision and tensor parallelism to align with our baselines.

Real Workloads. To model the bursty load patterns of online requests, we evaluated Concerto and the other baseline systems with a real-world load trace BurstGPT [263], which collects user requests to ChatGPT [189] and GPT-4 [190] in a university campus and is representative for online workloads. To adapt the campus-wide trace to our evaluation setting that consists of a relatively small number of GPUs, we sample the trace following the previous practice [230] as follows: given the duration D and request rate R, we sample $R \times D$ requests from the original trace, and re-scale the real-time stamps to $[0, D]$. We set R as the maximal request rate that can be served within latency SLOs. For offline workloads, we evaluated the document summarization task with the LongBench [23] datasets.

Synthetic Workloads. To demonstrate Concerto's capability in reacting to different request rates and different degrees of load burstiness and SLO tightness, we also leverage Concerto's load generator to generate synthetic workloads with configurable parameters (details in §6.6.3).

Metrics. Since online serving targets low latency and offline serving targets high throughput, we adopt different metrics when evaluating their service quality. For online serving, we measure each request's 99^{th} percentile TTFT and TPOT, respectively. For offline serving, we measure the throughput by counting the number of generated tokens per second.

6.6.2 Overall Serving Performance

In this section, we evaluated Concerto with real-world workloads and compared its end-to-end performance against the baselines. We used the same trace reported in Figure 6.1b as the online load. We first ran the original vLLM with only online loads to collect its 99^{th} percentile TTFT and TPOT, and then set them as the SLO targets for all systems (1500ms for TTFT and 110ms for TPOT).

A good result for Concerto would show that it quickly detects any GPU underutilization and judiciously batches offline requests to harvest available GPU resources to achieve good offline throughput while still keeping online serving latency lower than the SLO. In contrast, `Online-Only` should achieve the optimal online serving latency but fails to harvest idle GPU resources for offline serving. On the contrary, `vLLM++` does batch offline requests together with online ones, but it optimizes for the overall serving throughput and does not guarantee online latency. Therefore, we expect `vLLM++` to achieve high offline throughput but also experience drastically fluctuating TTFT and TPOT during load bursts.

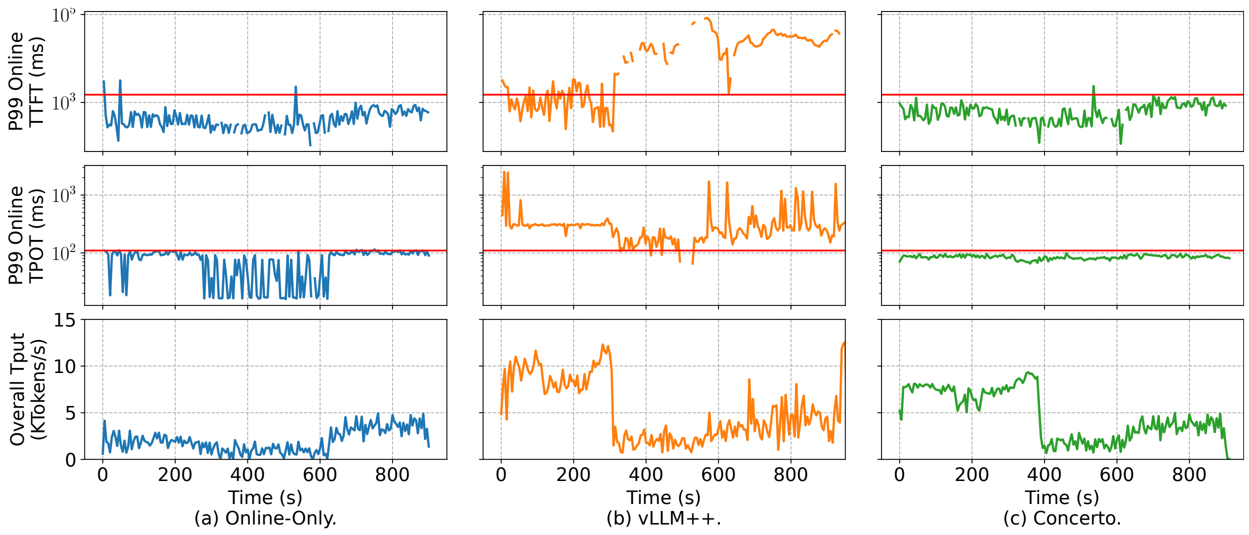


Figure 6.5: Overall serving performance on real workloads. Concerto achieves consistently low TTFT and TPOT that are comparable with `Online-Only` and below the SLO. It also achieves 86% of the ideal offline serving throughput (measured by `vLLM++` which eagerly batches offline requests regardless online latency constraints).

Figure 6.5 presents the results. The top two figures present 99th percentile TTFT and TPOT of all three systems. The bottom figure shows the offline serving throughput. Intuitively, `Online-Only` achieves optimal TTFT and TPOT. However, when the request rate is low (during time t = 240s-620s), it cannot batch enough requests per inference iteration, leading to overly low latency but GPU underutilization. On average, it achieves an overall serving throughput of 1999 tokens/s. Concerto maintains low online TTFT and TPOT that are close to ideal ones and consistently lower than SLOs, and it offers 3702 tokens/s overall throughput by harvesting available GPU resources for offline serving. `vLLM++` is also able to batch offline requests and achieves 4308 tokens/s throughput, but due to the frequent swapping, it ramps up the 99th percentile TTFT and TPOT by 84× (83825ms) and 25× (2523ms), respectively.

In summary, the experiment demonstrates that Concerto can efficiently utilize available GPU resources for offline serving with negligible impact on online serving latency. As a result,

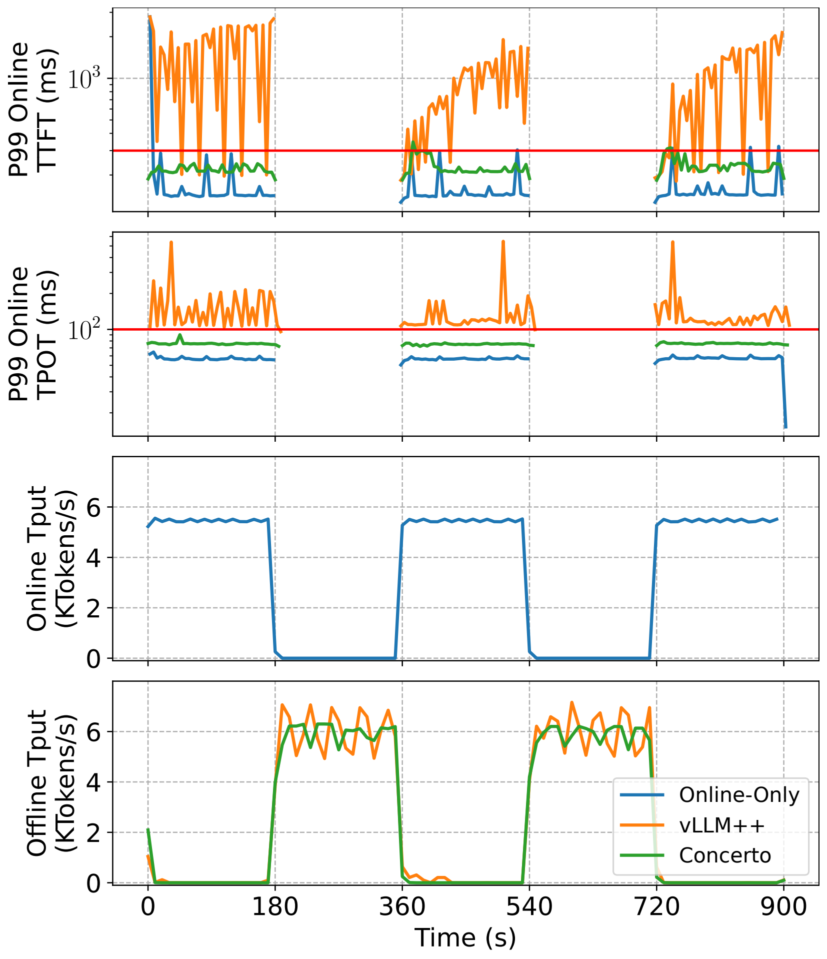


Figure 6.6: Concerto incurs negligible impact on online TTFT and TPOT and always keeps them below their SLO during ON phases. During the transition from the ON to OFF phase, Concerto reactively detects and harvests additional idle resources and achieves high offline throughput during the OFF phases. Even under extreme resource pressure during the transition from the OFF to ON phase, Concerto quickly scales down offline serving and prevents any spikes in online latency.

it achieves 2.35× higher overall throughput compared to `Online-Only` and 98.8% lower online serving latency compared to `vLLM++`.

6.6.3 Reacting to Load Bursts

In this section, we conducted a set of experiments to investigate whether Concerto can reactively harvest idle GPU resources—whenever they are available—for offline serving, while still quickly reacting to resource pressure to avoid interfering with online serving performance. For experiments in this section, we set the request input length to 1024 and the output length to 128 as representative values for online loads [263, 285].

6.6.3.1 ON/OFF Staged Load Patterns

In many real-world settings, LLMs do not always receive requests from clients. Instead, they may remain idle for a while ("OFF" phases), and experience high load occasionally ("ON" phases) [230]. To evaluate whether Concerto can react quickly enough to intense resource pressure when online bursts arrive, we synthesized a staged online load by dynamically changing the load between the system's max capacity and zero, as shown as the blue line in Figure 6.6. A good result for Concerto will show that it can quickly react to changes in resource availability and keep both low online tail latency and high GPU utilization.

Figure 6.6 presents the results. Initially, the system ran in the "ON" stage at its maximum capacity. At $t = 180$s, the system switched to the "OFF" stage with no online request until $t = 360$s, where the system started another round of the "ON" stage. Note that such resource pressure is extremely hard to handle because the load can spike instantly. Despite sharp changes between the "ON" and "OFF" stages, Concerto is still able to keep the 99th percentile TTFT and TPOT under 350ms and 90ms, respectively, and avoid SLO violations. Besides, it quickly and reactively regrants GPUs to offline serving at millisecond-scale when detecting idle GPU resources, thereby greatly improving GPU utilization and achieving an offline throughput of 6000 tokens/s during the "OFF" stages. `vLLM++`, in contrast, overly batches offline requests even in the "ON" stages, leading to 1.4× to 11× higher 99th percentile TTFT and TPOT and vast SLO violations.

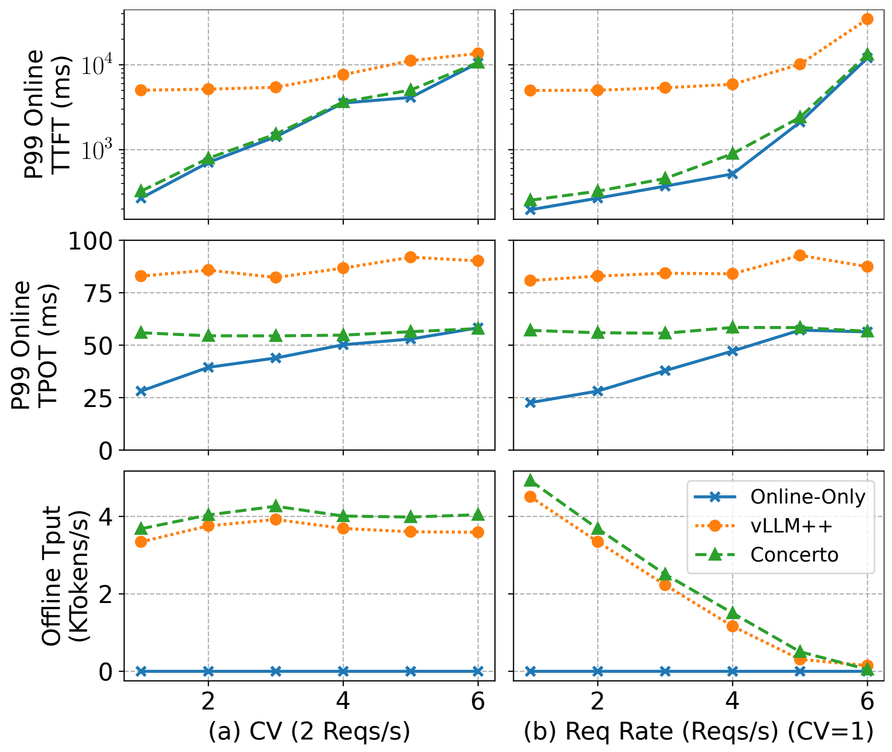


Figure 6.7: Overall serving performance under varying CVs and request rates. Concerto consistently achieves low online latencies and enables a linear trade-off between online throughput and offline throughput to keep maximized GPU utilization. It also maintains high efficiency across a wide range of load burstiness levels.

These results show that Concerto can quickly harvest idle GPU resources for offline serving and free them under pressure. Consequently, the online load neither runs out of resources nor slows down, and it only experiences negligible tail latency increases. This also means Concerto can keep GPUs at maximum utilization without any risk of interfering with online serving.

6.6.3.2 Robustness to Changing Load Burstiness

In reality, clients' behavior may have constantly changing load patterns [230, 263]. To this end, we further investigate the robustness of Concerto under varying load burstiness and request rates. Following previous studies [142, 263], we constructed a synthetic load following the Gamma process with an average rate of 2 requests per second and a coefficient of variation (CV) of 1. Here CV measures the load burstiness, and larger CV values mean that the load is more bursty. We evaluate how the online 99$^{\text{th}}$ percentile latencies change when fixing one factor and varying the other. As modeled by the queueing theory [66], we expect the queueing delay of requests will increase superlinearly as the CV or the request rate increases, but Concerto should be able to react to uncertain resource pressure and keep tail latencies comparable to the ideal ones.

Figure 6.7 presents 99$^{\text{th}}$ percentile online TTFTs, 99$^{\text{th}}$ percentile online TPOTs, and offline throughputs achieved by all three systems under varying CVs (left column) and varying request rates (right column), respectively. Intuitively, online TTFT increases when the load becomes more bursty and heavier for all systems. However, Concerto is robust to bursty loads and high request rates. It achieves low TTFTs that are close (within 25%) to the ideal latencies offered by `Online-Only`. `vLLM++`, in contrast, severely hurts online TTFTs with a minimum value of 4980ms. Such high 99$^{\text{th}}$ percentile TTFTs also mean `vLLM++` can hardly satisfy latency SLOs. Moreover, although Concerto typically batches fewer requests than `vLLM++` to keep latencies low, it can still outperform `vLLM++` in terms of offline serving throughput, and this is because Concerto overlaps checkpointing and swap-ins for offline requests and eliminates I/O stalls on GPUs.

In summary, these results illustrate that Concerto can always quickly react to intense resource pressure and avoid violating online latency SLOs, and it remains robust performance under varying request rates and load burstiness.

6.7 Discussion

Compatibility with Disaggregated LLM Serving Architectures. In latency-sensitive scenarios, users may have stringent TTFT and TPOT requirements. Therefore, many ongoing research efforts have proposed disaggregated architectures for LLM serving [198, 285] that use separate GPUs for prefill computation and decode computation to reduce inference. Concerto's design is orthogonal and compatible with these disaggregated architectures. Specifically, Concerto can be integrated separately in the prefill cluster scheduler and the decode cluster scheduler in a disaggregated architecture. Furthermore, Concerto's offline serving interfaces expose more semantics to reduce the KV cache transfer cost between prefill clusters and decode clusters. For example, Concerto can not only checkpoint to local DRAM but also asynchronously checkpoint offline KV cache from the prefill cluster to the GPU memory or DRAM in decode clusters. We leave these optimizations as future work.

Long-Context Scenarios. Improving LLMs' ability to process long contexts and long outputs has gained significant attraction [136, 145, 267]. Concerto is compatible with sequence parallelism and optimizations such as RingAttention [145] and StreamLLM [267] remain valid, in which case Concerto will partition both online and offline requests among all sequence parallel workers for load balancing.

Support for Multiple Models. While the main design goal of Concerto is to co-serve online and offline requests *within* a model to reuse the model weights, it can also be generalized to serve multiple models. For example, in practice, many models can share model weights but adapt to different tasks with parameter-efficient fine-tuning (PEFT) [45, 229, 266]. Concerto can seamlessly support them and flexibly co-serve online and offline requests for different fine-tuned models. For LLMs that do not share weights, Concerto can also serve them by co-locating multiple LLMs onto the same set of GPUs and routing requests to the target LLM.

6.8 Related Work

Model Serving Systems. Other than specialized LLM serving systems discussed in §6.2.3, many other systems target serving more general ML models. Triton [187], TorchServe [197, 243], TensorFlow Serving [9, 86] are three representative serving systems ready for production. Clipper [58], InferLine [59], and Clockwork [90] serve general neural networks by batching and scheduling requests. REEF [97] and SHEPHERD [277] proposed to co-locate models on the same GPU and preempt GPU compute kernels for scheduling. AlpaServe [142], on the other hand, leverages model parallelism for statistical multiplexing. However, these systems overlook the huge model size and the autoregressive nature of LLM inference, hence only achieving suboptimal LLM serving performance.

Offline LLM Serving. As offline inference has gained increasing traction, many systems are specifically optimized for offline LLM serving. DeepSpeed ZeRO-Inference [17] and FlexGen [228] proposed to offload model weights and KV caches to host memory to serve LLMs on small commodity GPUs. Their design does not fit online serving due to the long swapping latency, but they can also benefit from Concerto's incremental checkpointing mechanism for further performance improvement. S^3 [116] optimized for high generation throughput by predicting the output sequence length and minimizing memory waste. Concerto is orthogonal to these optimizations and can further benefit from them for higher offline serving throughput.

Optimized LLM Algorithms. Another line of work focuses on optimizing the efficiency of LLM algorithms/kernels. FlashAttention [62, 63] improves the memory I/O efficiency with a redesigned attention kernel. GPTQ [75], AWQ [144], and SqueezeLLM [123] quantize and compress the model weights and KV caches to reduce GPU memory consumption. Some other work aims to improve compute and memory efficiency with optimized transformer architectures. MQA [226] and GQA [11] modify the attention kernel to reduce the KV cache size. Mixture-of-expert models [21, 114, 166] make weight parameters sparse and hence reduce the model size. Concerto is orthogonal to these optimizations on algorithms and architectures, while Concerto can further improve GPU utilization beyond their benefits.

Deep Learning Schedulers. GPU clusters today suffer from low resource utilization [111, 264, 269]. To improve GPU utilization, many deep learning schedulers are proposed for better model placement, job migration, and GPU sharing [264, 265, 268, 269, 273]. Concerto focuses on sharing GPU resources between online and offline serving, and it is orthogonal to cluster-level schedulers. However, Concerto could benefit from advances in underlying scheduling policies and hardware support.

Workload Co-location. Co-locating latency-critical applications with batch applications is a widely adopted approach to improve resource utilization in datacenters. For instance, Parties [46] partitions resources such as CPU cache and memory resources across microservices to preserve their SLOs. Operating systems such as ZygOS [204], Shenango [194], and Caladan [76] proposed to preempt batch jobs and reallocate CPU cores to latency-critical jobs to improve CPU utilization. However, existing workload co-location solutions primarily target traditional hardware resources and workloads, rather than emerging GPUs and AI workloads. In contrast, Concerto shares the concept of colocating workloads and preemption to improve resource utilization but is specifically optimized for LLM serving on GPUs.

6.9 Summary

In this chapter, we present Concerto, a unified LLM serving system that serves both online and offline inference with near-optimal efficiency and GPU utilization. Concerto achieves these benefits through its preemptive worker, novel incremental checkpointing mechanism, and adaptive scheduler, which opportunistically schedules offline requests when possible, and timely preempts them before they can impact online serving latency. Concerto proves that it is possible to maintain low latency, high throughput, and high GPU utilization simultaneously.

CHAPTER 7

Conclusion

Today's datacenters struggle with low resource utilization, primarily due to a mismatch between the modern applications they host—characterized by high load variability and strict performance requirements—and outdated system stacks that fail to efficiently harness idle resources. In this dissertation, we present a vision and a comprehensive, four-part approach to redesigning datacenter operating systems and runtime systems, significantly improving their efficiency and safety for resource harvesting.

Our redesigned datacenter system stacks are grounded in two key insights. First, although datacenter applications often exhibit variable and large resource demands, many include elastic components that can utilize idle resources with intermittent availability. Second, aggregating resources across servers can form a more stable and sufficient resource pool, even when resource availability per server is unpredictable.

Based on these insights, this dissertation introduces four resource harvesting systems. First, Chapter 3 presents Midas, a soft memory management system that harvests memory within a server. To bridge the semantics gap between the application and the operating system, we propose soft memory, a new OS memory abstraction for application-managed soft state. Soft memory captures the essential unmap-and-reconstruct semantics, enabling the runtime and the OS kernel to co-manage idle memory with guaranteed safety and efficiency.

Expanding beyond a single server, Chapters 4 and 5 explore systems that enable applications to harness idle memory on a remote server. In particular, Chapter 4 introduces Hermit, a fully asynchronous OS kernel swap system for remote memory that allows applications to transparently and efficiently scale out by swapping their data to remote memory. Further, Chapter 5 presents Canvas, which enhances the OS kernel swap system with isolation support and adaptive applications. Canvas enables multiple applications to share remote memory simultaneously without performance interference, moving a step closer to the practical deployment of remote memory.

Lastly, we extend these insights to encompass emerging datacenter hardware and workloads, exemplified by GPUs and large language models. Chapter 6 presents Concerto, a preemptive GPU runtime designed for large language model serving, which effectively harvests idle GPU resources for offline inference tasks.

Collectively, these systems form a holistic system stack for resource harvesting, demonstrating the feasibility of safely and efficiently utilizing all available resources in datacenters, including both traditional resources like memory and accelerators like GPUs, within and across servers. This dissertation not only addresses the pressing issue of resource underutilization in datacenters but also lays the foundation for more adaptive, resource-efficient infrastructures in the future.

7.1 Future Directions

Looking ahead, we believe that the full potential of resource harvesting can be realized through more comprehensive system stack support, building upon the foundations laid in this dissertation. In this section, we outline several promising directions for future work that will push the boundaries of what is possible in datacenter resource management. As illustrated in Figure 7.1, these directions include building accelerator-centric system stacks for AI/ML workloads, developing programming frameworks for resource harvesting, and abstracting heterogeneous and distributed hardware resources as unified services.

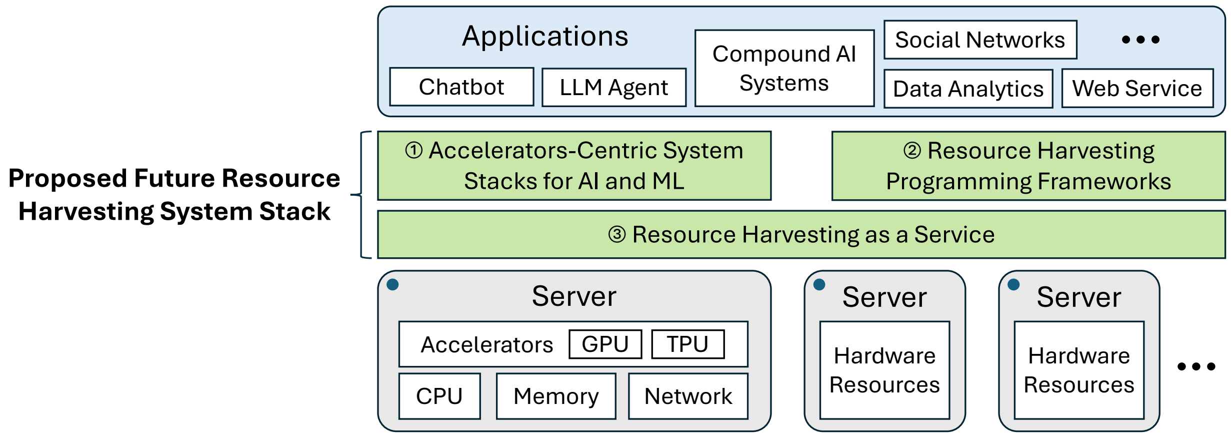

Applications
Chatbot
LLM Agent
Compound AI Systems
Social Networks
Data Analytics
Web Service
Proposed Future Resource Harvesting System Stack
① Accelerators-Centric System Stacks for AI and ML
② Resource Harvesting Programming Frameworks
③ Resource Harvesting as a Service
Server
Accelerators
GPU
TPU
CPU
Memory
Network
Server
Hardware Resources
Server
Hardware Resources

Figure 7.1: Overview of the proposed future resource harvesting system stack, comprising three major components. This stack is designed to support a broad range of applications, from traditional datacenter workloads to emerging AI workloads, while efficiently managing heterogeneous hardware resources through unified abstractions and flexible programming interfaces.

Accelerator-Centric System Stacks for AI and ML. GPUs and other accelerators are evolving into the new central processing units in datacenters, powering numerous deep-learning and AI workloads. Meanwhile, emerging AI applications such as AI agents are not only growing in scale but also in complexity. These applications start to integrate with multiple ML models and other software tools, each with diverse resource demands [130, 148, 191, 201, 275, 280]. However, due to the lack of comprehensive operating system support for GPUs, AI applications today can only leverage vendor-specific drivers and runtimes, such as CUDA [188], to use GPUs, which only offer primitive support for resource management. For instance, these drivers and runtimes do not support distributing tasks over multiple GPUs, nor flexibly scheduling multiple tasks onto and time-sharing a single GPU. As a result, they have to build their own runtime systems and manage resources manually in a coarse granularity, suffering from unnecessary complexity and inefficiency that could have been avoided with better system stack support.

To address this, we propose that, akin to how traditional operating systems virtualize

CPU cores and physical memory, we should introduce similar abstractions to accelerator resources. Our proposed system stack will abstract physical resources, allowing applications to interact with high-level interfaces rather than dealing with hardware details directly. This system will offer enhanced scheduling support, enabling multiple applications to share the same GPU, and provide autoscaling capabilities to quickly launch or migrate jobs across GPUs, potentially across different servers.

From the user's perspective, the benefit of having an OS and runtime can be further amplified because they can build applications easier with high-level interfaces. For example, we propose to build new programming frameworks that encapsulate low-level abstractions we just mentioned for developers to build AI applications that are distributed and heterogeneous and may have multiple components that use both CPUs and GPUs.

Resource Harvesting Programming Frameworks. Today's datacenter applications are constructed upon *layers of abstractions*, encompassing programming languages, application runtimes, VMs and the OS kernel, and hardware. While this paradigm reduces programming effort, it often sacrifices efficiency because abstractions conceal application semantics, thereby limiting opportunities to customize the underlying systems. This inefficiency can be addressed through a holistic design spanning multiple system layers, particularly through co-designing programming languages and systems to reshape future cloud applications with semantics-aware system support. For instance, Golang's reliance on coroutines naturally segments programs into finer-grained pieces [4], offering opportunities for distributed scheduling optimization to improve scalability and resource efficiency. Similarly, Rust's ownership memory model [8, 153], which ensures exclusive write access to shared memory objects, can be harnessed for efficient data distribution and sharing and providing applications with a distributed shared memory without incurring expensive coherence and synchronization overheads. By leveraging the hidden semantics of existing language abstractions and introducing new abstractions (as demonstrated in Canvas and Midas), we can build a new programming framework that empowers developers to write applications as logical monoliths while offloading deployment tasks to the runtime, which can automatically distribute applications across servers with high scalability, resource efficiency, and fault resilience.

Resource Harvesting as a Service. Microservice-based applications demand extensive caching, often involving complex dependencies between cached objects [276]. For example, in a Twitter-like web application, the frontend service might cache user posts for quicker timeline rendering, with each post potentially containing several media objects cached by different microservices [80]. The typical approach—serializing these objects along with their dependencies into a key-value cache—results in memory waste due to object duplication and complicates cache coherence when multiple copies of the same object exist.

While soft memory pointers in Midas allow developers to nest soft pointers to express dependencies between objects, it currently manages soft objects on a per-process and per-server basis, hindering its adoption in distributed microservices. To overcome this limitation, we propose developing a distributed cache service that allows processes to harvest idle memory from any server for storing soft objects, acquire cross-server soft pointers, and share soft objects efficiently. Achieving this goal will require a co-design of the application runtime and OS kernel. The kernel will expose shared soft memory with a new virtual memory abstraction, manage the application's permission for safety, and leverage remote memory support for cross-server accesses, while the application runtime will coordinate with the kernel for cache entry allocation and eviction. It will also provide high-level APIs like pass-by-reference RPCs to facilitate microservices in using the cache service and harvesting available compute resources on remote servers.

Bibliography

[1] Amazon elasticache. https://aws.amazon.com/elasticache/, 2023.

[2] The redis database. https://redis.com/, 2023.

[3] Storage performance development kit. https://spdk.io, 2023.

[4] Goroutines. https://go.dev/tour/concurrency/1, 2024. URL https://go.dev/tour/concurrency/1.

[5] HotSpot Glossary of Terms: Safepoint. https://openjdk.org/groups/hotspot/docs/HotSpotGlossary.html, 2024. URL https://openjdk.org/groups/hotspot/docs/HotSpotGlossary.html.

[6] System Management Interface SMI. https://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf, 2024. URL https://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf.

[7] gc—Garbage Collector interface. https://docs.python.org/3/library/gc.html, 2024. URL https://docs.python.org/3/library/gc.html.

[8] Rust onwership model. https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html, 2024. URL https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html.

[9] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[10] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in llm inference with sarathi-serve, 2024.

[11] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints, 2023.

[12] 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 *EuroSys*, 2020.

[13] Emmanuel Amaro, Zhihong Luo, Amy Ousterhout, Arvind Krishnamurthy, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Remote memory calls. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, HotNets '20, page 38–44, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381451. doi: 10.1145/3422604.3425923. URL https://doi.org/10.1145/3422604.3425923.

[14] Emmanuel Amaro, Stephanie Wang, Aurojit Panda, and Marcos K. Aguilera. Logical memory pools: Flexible and local disaggregated memory. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, HotNets '23, page 25–32, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400704154. doi: 10.1145/3626111.3628201. URL https://doi.org/10.1145/3626111.3628201.

[15] Amazon Elastic Compute Cloud. Amazon ec2 spot instances. https://aws.amazon.com/ec2/spot, 2022.

[16] Pradeep Ambati, Inigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. Providing SLOs for Resource-Harvesting VMs in cloud platforms. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 735–751. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/ambati.

[17] Reza Yazdani Aminabadi, Samyam Rajbhandari, Minjia Zhang, Ammar Ahmad Awan,

Cheng Li, Du Li, Elton Zheng, Jeff Rasley, Shaden Smith, Olatunji Ruwase, and Yuxiong He. Deepspeed inference: Enabling efficient inference of transformer models at unprecedented scale, 2022. URL https://arxiv.org/abs/2207.00032.

[18] Apache Cassandra. An open source nosql database. https://cassandra.apache.org, 2021.

[19] Dan Ardelean, Amer Diwan, and Chandra Erdman. Performance analysis of cloud applications. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 405–417, Renton, WA, April 2018. USENIX Association. ISBN 978-1-939133-01-4. URL https://www.usenix.org/conference/nsdi18/presentation/ardelean.

[20] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, Feb 2009. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html.

[21] Mikel Artetxe, Shruti Bhosale, Naman Goyal, Todor Mihaylov, Myle Ott, Sam Shleifer, Xi Victoria Lin, Jingfei Du, Srinivasan Iyer, Ramakanth Pasunuru, Giri Anantharaman, Xian Li, Shuohui Chen, Halil Akin, Mandeep Baines, Louis Martin, Xing Zhou, Punit Singh Koura, Brian O'Horo, Jeff Wang, Luke Zettlemoyer, Mona Diab, Zornitsa Kozareva, and Ves Stoyanov. Efficient large scale language modeling with mixtures of experts, 2022.

[22] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Asso-

ciation for Computing Machinery. ISBN 9781450310970. doi: 10.1145/2254756.2254766. URL https://doi.org/10.1145/2254756.2254766.

[23] Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. Longbench: A bilingual, multitask benchmark for long context understanding, 2024. URL https://arxiv.org/abs/2308.14508.

[24] Pierre Baldi, Peter Sadowski, and Daniel Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature communications*, 5(1):1–9, 2014.

[25] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards predictable datacenter networks. In *SIGCOMM*, pages 242–253, 2011.

[26] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, 2017.

[27] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. *The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition*. Synthesis Lectures on Computer Architecture, 2018.

[28] Davide B. Bartolini, Filippo Sironi, Donatella Sciuto, and Marco D. Santambrogio. Automated fine-grained cpu provisioning for virtual machines. *ACM Trans. Archit. Code Optim.*, 11(3), July 2014.

[29] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *OSDI*, pages 49–65, 2014.

[30] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. The CacheLib caching engine: Design and experiences at scale. In

14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pages 753–768. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/berg.

[31] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. RobinHood: Tail latency aware caching – dynamic reallocation from Cache-Rich to Cache-Poor. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 195–212, Carlsbad, CA, October 2018. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/berger.

[32] Dimitri Bertsekas and Robert Gallager. *Data Networks (2nd Ed.)*. Prentice-Hall, Inc., USA, 1992. ISBN 0132009161.

[33] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site reliability engineering: How Google runs production systems.* " O'Reilly Media, Inc.", 2016.

[34] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient distributed memory management with rdma and caching. *Proceedings of the VLDB Endowment*, 11(11): 1604–1617, 2018.

[35] Irina Calciu, Ivan Puddu, Aasheesh Kolli, Andreas Nowatzyk, Jayneel Gandhi, Onur Mutlu, and Pratap Subrahmanyam. Project pberry: Fpga acceleration for remote memory. In *HotOS*, HotOS '19, pages 127–135, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367271.

[36] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. *Rethinking Software Runtimes for Disaggregated Memory*, page 79–92. Association for Computing Machinery, New York, NY, USA, 2021. ISBN 9781450383172. URL https://doi.org/10.1145/3445814.3446713.

[37] Matt Calder, Ryan Gao, Manuel Schröder, Ryan Stewart, Jitendra Padhye, Ratul Mahajan, Ganesh Ananthanarayanan, and Ethan Katz-Bassett. Odin: Microsoft's scalable Fault-Tolerant CDN measurement system. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 501–517, Renton, WA, April 2018. USENIX Association. ISBN 978-1-939133-01-4. URL https://www.usenix.org/conference/nsdi18/presentation/calder.

[38] Roy Campbell, Garry Johnston, and Vincent Russo. Choices (class hierarchical open interface for custom embedded systems). *ACM SIGOPS Operating Systems Review*, 21 (3):9–17, 1987.

[39] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Trans. Comput. Syst.*, 14(4):311–343, November 1996.

[40] John B Carter, John K Bennett, and Willy Zwaenepoel. Implementation and performance of munin. *ACM SIGOPS Operating Systems Review*, 25(5):152–164, 1991.

[41] John B Carter, John K Bennett, and Willy Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared-memory systems. *ACM Transactions on Computer Systems (TOCS)*, 13(3):205–243, 1995.

[42] CBRE. North america data center trends h2 2021. https://www.cbre.com/insights/reports/north-america-data-center-trends-h2-2021, 2022.

[43] Chih-Chung Chang and Chih-Jen Lin. Libsvm data: Classification. https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets, 2012.

[44] Lei Chen, Shi Liu, Chenxi Wang, Haoran Ma, Yifan Qiao, Zhe Wang, Chenggang Wu, Youyou Lu, Xiaobing Feng, Huimin Cui, Shan Lu, and Harry Xu. A tale of two paths: Toward a hybrid data plane for efficient Far-Memory applications. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 77–95,

Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/chen-lei.

[45] Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. Punica: Multi-tenant lora serving, 2023.

[46] Shuang Chen, Christina Delimitrou, and José F. Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 107–120, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362405. doi: 10.1145/3297858.3304005. URL https://doi.org/10.1145/3297858.3304005.

[47] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *KDD*, pages 785–794, 2016.

[48] Tianqi Chen and Carlos Guestrin. extreme gradient boosting for applied machine learning. https://xgboost.readthedocs.io/en/latest/, 2021.

[49] Youmin Chen, Youyou Lu, and Jiwu Shu. Scalable RDMA RPC on reliable connection with efficient resource sharing. In *EuroSys*, 2019.

[50] Ludmila Cherkasova, Diwaker Gupta, and Amin Vahdat. Comparison of the three cpu schedulers in xen. *SIGMETRICS Perform. Eval. Rev.*, 35(2):42–51, 2007.

[51] Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Hao Zhang, Banghua Zhu, Michael Jordan, Joseph E. Gonzalez, and Ion Stoica. Chatbot arena: An open platform for evaluating llms by human preference, 2024.

[52] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on*

Networked Systems Design and Implementation (NSDI 16), pages 379–392, Santa Clara, CA, March 2016. USENIX Association. ISBN 978-1-931971-29-4. URL https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/cidon.

[53] Asaf Cidon, Daniel Rushton, Stephen M. Rumble, and Ryan Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 321–334, Santa Clara, CA, July 2017. USENIX Association. ISBN 978-1-931971-38-6. URL https://www.usenix.org/conference/atc17/technical-sessions/presentation/cidon.

[54] Adrian Cockroft. The Evolution of Microservices. https://www.slideshare.net/slideshow/evolution-of-microservices-craft-conference/61466608, 2024. URL https://www.slideshare.net/slideshow/evolution-of-microservices-craft-conference/61466608.

[55] Adrian Cockroft. Microservice Workshop: Why, What, and How to Get There. https://www.slideshare.net/slideshow/microservices-workshop-craft-conference/61466758, 2024. URL https://www.slideshare.net/slideshow/microservices-workshop-craft-conference/61466758.

[56] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300360. doi: 10.1145/1807128.1807152. URL https://doi.org/10.1145/1807128.1807152.

[57] F.J. Corbató. *A Paging Experiment With the MULTICS System*. Project MAC. Massachusetts Institute of Technology, 1968. URL https://books.google.com/books?id=5wDQNwAACAAJ.

[58] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez,

and Ion Stoica. Clipper: A Low-Latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, Boston, MA, March 2017. USENIX Association. ISBN 978-1-931971-37-9. URL https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw.

[59] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. Inferline: latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 477–491, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381376. doi: 10.1145/3419111.3421285. URL https://doi.org/10.1145/3419111.3421285.

[60] Bryce Cronkite-Ratcliff, Aran Bergman, Shay Vargaftik, Madhusudhan Ravi, Nick McKeown, Ittai Abraham, and Isaac Keslassy. Virtualized congestion control. In *SIGCOMM*, pages 230–243, 2016.

[61] cxl. Compute express link 3.0. https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-3.0-Specification.pdf, 2022.

[62] Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations (ICLR)*, 2024.

[63] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[64] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56:74–80, 2013. URL http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext.

[65] Christina Delimitrou and Christos Kozyrakis. Bolt: I know what you did last summer... in the cloud. In *ASPLOS*, pages 599–613, 2017.

[66] Christina Delimitrou and Christos Kozyrakis. Amdahl's law for tail latency. *Commun. ACM*, 61(8):65–72, jul 2018. ISSN 0001-0782. doi: 10.1145/3232559. URL https://doi.org/10.1145/3232559.

[67] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *SIGCOMM Comput. Commun. Rev.*, 19(4):1–12, August 1989. ISSN 0146-4833. doi: 10.1145/75247.75248. URL https://doi.org/10.1145/75247.75248.

[68] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *NSDI*, pages 401–414, 2014.

[69] Facebook and Intel. Facebook and intel collaborate on future data center rack technologies. http://goo.gl/6h2Ut, 2013.

[70] Michael Ferdman, Cansu Kaynak, and Babak Falsafi. Proactive instruction fetch. In *MICRO*, pages 152–162, 2011.

[71] Brett D Fleisch. Distributed shared memory in a loosely coupled distributed system. *ACM SIGCOMM Computer Communication Review*, 17(5):317–327, 1987.

[72] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993. doi: 10.1109/90.251892.

[73] Linux Foundation. Data plane development kit (DPDK). http://www.dpdk.org, 2015. URL http://www.dpdk.org.

[74] The OCP Foundation. Multi-generational lru: the next generation. https://lwn.net/Articles/856931/.

[75] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers, 2023.

[76] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *OSDI*. USENIX Association, 2020. URL https://www.usenix.org/conference/osdi20/presentation/fried.

[77] Megan Frisella, Shirley Loayza Sanchez, and Malte Schwarzkopf. Towards increased datacenter efficiency with soft memory. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HOTOS '23, page 127–134, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701955. doi: 10.1145/3593856.3595902. URL https://doi.org/10.1145/3593856.3595902.

[78] Alexander Fuerst, Stanko Novaković, Íñigo Goiri, Gohar Irfan Chaudhry, Prateek Sharma, Kapil Arya, Kevin Broas, Eugene Bak, Mehmet Iyigun, and Ricardo Bianchini. Memory-harvesting vms in cloud platforms. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 583–594, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392051. doi: 10.1145/3503222.3507725. URL https://doi.org/10.1145/3503222.3507725.

[79] Liran Funaro, Orna Agmon Ben-Yehuda, and Assaf Schuster. Ginseng: Market-driven LLC allocation. In *USENIX ATC*, pages 295–308, 2016.

[80] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[81] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *OSDI*, pages 249–264, 2016.

[82] gdnsd. An authoritative-only dns server. https://gdnsd.org/.

[83] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, pages 323–336, 2011.

[84] GitHub. Github copilot - write code faster. https://copilot.github.com/, 2021.

[85] Google. Google's fast compressor/decompressor. https://github.com/google/snappy, 2020.

[86] Google. Tensorflow serving is a flexible, high-performance serving system for machine learning models. https://www.tensorflow.org/tfx/guide/serving, 2024.

[87] Google Cloud. Preemptible vm instances. https://cloud.google.com/compute/docs/instances/preemptible, 2022.

[88] Grand View Research Inc. Servers market share. https://www.grandviewresearch.com/industry-analysis/server-market, 2022.

[89] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with infiniswap. In *NSDI*, pages 649–667, 2017.

[90] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/gujarati.

[91] Ajay Gulati, Arif Merchant, and Peter J. Varman. MClock: Handling throughput variability for hypervisor IO scheduling. In *OSDI*, pages 437–450, 2010.

[92] Chuanxiong Guo, Guohan Lu, Helen J. Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. SecondNet: A data center network virtualization architecture with bandwidth guarantees. In *Co-NEXT*, 2010.

[93] Yao Guo. *Compiler-Assisted Hardware-Based Data Prefetching for Next Generation Processors*. PhD thesis, University of Massachusetts Amherst, 2007.

[94] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2022, page 417–433, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392051. doi: 10.1145/3503222.3507762. URL https://doi.org/10.1145/3503222.3507762.

[95] Zhiyuan Guo, Zijian He, and Yiying Zhang. Mira: A program-behavior-guided far memory system. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 692–708, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613157. URL https://doi.org/10.1145/3600006.3613157.

[96] Zhiyuan Guo, Zachary Blanco, Junda Chen, Jinmou Li, Zerui Wei, Bili Dong, Ishaan Pota, Mohammad Shahrad, Harry Xu, and Yiying Zhang. Zenix: Efficient execution of bulky serverless applications, 2024. URL https://arxiv.org/abs/2206.13444.

[97] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 539–558, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/han.

[98] Keqiang He, Eric Rozner, Kanak Agarwal, Yu (Jason) Gu, Wes Felter, John Carter,

and Aditya Akella. AC/DC TCP: Virtual congestion control enforcement for datacenter networks. In *SIGCOMM*, pages 244–257, 2016.

[99] Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, and Yuxiong He. Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference, 2024.

[100] Liam Howlett and Matthew Wilcox. Introducing maple trees. https://lwn.net/Articles/845507/.

[101] Krystal Hu. Chatgpt sets record for fastest-growing user base. https://www.reuters.com/technology/chatgpt-sets-record-fastest-growing-user-base-analyst-note-2023-02-01/, 2023.

[102] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. TCP $\approx$ RDMA: CPU-efficient remote storage access with i10. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 127–140, Santa Clara, CA, February 2020. USENIX Association. ISBN 978-1-939133-13-7. URL https://www.usenix.org/conference/nsdi20/presentation/hwang.

[103] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanopu: A nanosecond network stack for datacenters. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 239–256. USENIX Association, July 2021. ISBN 978-1-939133-22-9. URL https://www.usenix.org/conference/osdi21/presentation/ibanez.

[104] IDC Corporate. Servers market share. https://www.idc.com/promo/servers, 2022.

[105] Intel. Memcontrol: Charge swap-in pages to `cgroup`. https://github.com/torvalds/linux/commit/4c6355b25e8bb83c, 2020.

[106] Calin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. PerfIso: Performance isolation for commercial Latency-Sensitive services. In *USENIX ATC*, pages 519–532, 2018.

[107] 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 *SIGMETRICS*, pages 25–36, 2007.

[108] Jacob Leverich. Mutilate: High-performance memcached load generator. `https://github.com/leverich/mutilate`, 2023.

[109] Akanksha Jain and Calvin Lin. Linearizing irregular memory accesses for improved correlated prefetching. In *MICRO*, pages 247–259, 2013.

[110] Naman Jain, Tianjun Zhang, Wei-Lin Chiang, Joseph E. Gonzalez, Koushik Sen, and Ion Stoica. Llm-assisted code cleaning for training accurate code generators, 2023.

[111] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of Large-Scale Multi-Tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, Renton, WA, July 2019. USENIX Association. ISBN 978-1-939133-03-8. URL `https://www.usenix.org/conference/atc19/presentation/jeon`.

[112] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. MTCP: A highly scalable user-level TCP stack for multicore systems. In *NSDI*, pages 489–502, 2014.

[113] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, and Changhoon Kim. EyeQ: Practical network performance isolation for the multi-tenant cloud. In *HotCloud*, 2012.

[114] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mixtral of experts, 2024.

[115] Hanlei Jin, Yang Zhang, Dan Meng, Jun Wang, and Jinghua Tan. A comprehensive survey on process-oriented automatic text summarization with exploration of llm-based methods, 2024. URL https://arxiv.org/abs/2403.02901.

[116] Yunho Jin, Chun-Feng Wu, David Brooks, and Gu-Yeon Wei. S3: increasing gpu utilization during generative inference for higher throughput. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2024. Curran Associates Inc.

[117] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *ISCA*, pages 252–263, 1997.

[118] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *OSDI*, pages 185–201, 2016.

[119] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance RDMA systems. In *USENIX ATC*, pages 437–450, 2016.

[120] Harshad Kasture and Daniel Sanchez. Ubik: Efficient cache sharing with strict qos for latency-critical workloads. In *ASPLOS*, pages 729–742, 2014.

[121] Stefanos Kaxiras, David Klaftenegger, Magnus Norgren, Alberto Ros, and Konstantinos Sagonas. Turning centralized coherence and distributed critical-section execution on their head: A new approach for scalable distributed shared memory. In *Proceedings*

of the 24th International Symposium on High-Performance Parallel and Distributed Computing, pages 3–14, 2015.

[122] Junaid Khalid, Eric Rozner, Wesley Felter, Cong Xu, Karthick Rajamani, Alexandre Ferreira, and Aditya Akella. Iron: Isolating network-based cpu in container environments. In *NSDI*, pages 313–328, 2018.

[123] Sehoon Kim, Coleman Hooper, Amir Gholami, Zhen Dong, Xiuyu Li, Sheng Shen, Michael W. Mahoney, and Kurt Keutzer. Squeezellm: Dense-and-sparse quantization, 2024.

[124] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. ReFlex: Remote flash ≈ local flash. In *ASPLOS*, pages 345–359, 2017.

[125] A. Kolli, A. Saidi, and T. F. Wenisch. RDIP: Return-address-stack directed instruction prefetching. In *MICRO*, pages 260–271, 2013.

[126] Jérôme Kunegis. Wikipedia networks data. http://konect.uni-koblenz.de/networks/, 2020.

[127] Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Alexander Kolesnikov, Tom Duerig, and Vittorio Ferrari. The open images dataset v4: Unified image classification, object detection, and visual relationship detection at scale. *IJCV*, 2020.

[128] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

[129] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. Software-defined far memory in warehouse-scale computers. In *ASPLOS*, pages 317–330, 2019.

[130] LangChain. Langchain: Build context-aware reasoning application. https://python.langchain.com/, 2024.

[131] Chris Lattner and Vikram Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *PLDI*, pages 129–142, 2005.

[132] Seok-Hee Lee. Technology scaling challenges and opportunities of memory devices. In *2016 IEEE International Electron Devices Meeting (IEDM)*, pages 1.1.1–1.1.8, 2016. doi: 10.1109/IEDM.2016.7838026.

[133] Sergey Legtchenko, Hugh Williams, Kaveh Razavi, Austin Donnelly, Richard Black, Andrew Douglas, Nathanaël Cheriere, Daniel Fryer, Kai Mast, Angela Demke Brown, Ana Klimovic, Andy Slowey, and Antony Rowstron. Understanding rack-scale disaggregated storage. In *HotStorage*, 2017.

[134] Michel Lespinasse. Speculative page faults. https://lwn.net/Articles/851853/.

[135] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 137–152, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350853. doi: 10.1145/3132747.3132756. URL https://doi.org/10.1145/3132747.3132756.

[136] Dacheng Li, Rulin Shao, Anze Xie, Eric P. Xing, Xuezhe Ma, Ion Stoica, Joseph E.

Gonzalez, and Hao Zhang. Distflashattn: Distributed memory-efficient attention for long-context llms training, 2024. URL https://arxiv.org/abs/2310.03294.

[137] Huaicheng Li, Daniel S. Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. First-generation memory disaggregation for cloud platforms, 2022. URL https://arxiv.org/abs/2203.00241.

[138] Kai Li. Ivy: A shared virtual memory system for parallel computing. *ICPP (2)*, 88:94, 1988.

[139] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, November 1989. ISSN 0734-2071.

[140] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, 1989.

[141] Tong Li, Dan Baumberger, and Scott Hahn. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In *PPoPP*, pages 65–74, 2009.

[142] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679, Boston, MA, July 2023. USENIX Association. ISBN 978-1-939133-34-2. URL https://www.usenix.org/conference/osdi23/presentation/li-zhouhan.

[143] Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, Benjamin Newman, Binhang Yuan, Bobby Yan, Ce Zhang, Christian Cosgrove, Christopher D. Manning, Christopher Ré, Diana Acosta-Navas, Drew A. Hudson, Eric Zelikman, Esin Durmus, Faisal Ladhak, Frieda Rong, Hongyu Ren, Huaxiu Yao, Jue Wang, Keshav

Santhanam, Laurel Orr, Lucia Zheng, Mert Yuksekgonul, Mirac Suzgun, Nathan Kim, Neel Guha, Niladri Chatterji, Omar Khattab, Peter Henderson, Qian Huang, Ryan Chi, Sang Michael Xie, Shibani Santurkar, Surya Ganguli, Tatsunori Hashimoto, Thomas Icard, Tianyi Zhang, Vishrav Chaudhary, William Wang, Xuechen Li, Yifan Mai, Yuhui Zhang, and Yuta Koreeda. Holistic evaluation of language models, 2023. URL https://arxiv.org/abs/2211.09110.

[144] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for llm compression and acceleration, 2024.

[145] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context, 2023. URL https://arxiv.org/abs/2310.01889.

[146] Lei Liu, Zehan Cui, Mingjie Xing, Yungang Bao, Mingyu Chen, and Chengyong Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *PACT*, pages 367–376, 2012.

[147] Lei Liu, Yong Li, Zehan Cui, Yungang Bao, Mingyu Chen, and Chengyong Wu. Going vertical in memory management: Handling multiplicity by multi-policy. In *ISCA*, pages 169–180, 2014.

[148] Shu Liu, Asim Biswal, Audrey Cheng, Xiangxi Mo, Shiyi Cao, Joseph E. Gonzalez, Ion Stoica, and Matei Zaharia. Optimizing llm queries in relational workloads, 2024.

[149] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Improving resource efficiency at scale with heracles. *ACM Trans. Comput. Syst.*, 34(2), 2016.

[150] C. Lu, K. Ye, G. Xu, C. Xu, and T. Bai. Imbalance in the cloud: An analysis on Alibaba cluster trace. In *Big Data*, pages 2884 – 2892, 2017.

[151] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Physical disentanglement in a container-based file system. In *OSDI*, pages 81–96, 2014.

[152] Haoran Ma, Shi Liu, Chenxi Wang, Yifan Qiao, Michael D. Bond, Stephen M. Blackburn, Miryung Kim, and Guoqing Harry Xu. Mako: A low-pause, high-throughput evacuating collector for memory-disaggregated datacenters. In *PLDI*, 2022.

[153] Haoran Ma, Yifan Qiao, Shi Liu, Shan Yu, Yuanjiang Ni, Qingda Lu, Jiesheng Wu, Yiying Zhang, Miryung Kim, and Harry Xu. DRust: Language-Guided distributed shared memory with fine granularity, full transparency, and ultra efficiency. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 97–115, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/ma-haoran.

[154] Jiuyue Ma, Xiufeng Sui, Ninghui Sun, Yupeng Li, Zihao Yu, Bowen Huang, Tianni Xu, Zhicheng Yao, Yun Chen, Haibin Wang, Lixin Zhang, and Yungang Bao. Supporting differentiated services in computers via programmable architecture for resourcing-on-demand (PARD). In *ASPLOS*, pages 131–143, 2015.

[155] Martin Maas, Krste Asanović, Tim Harris, and John Kubiatowicz. Taurus: A holistic language runtime system for coordinating distributed managed-language applications. In *ASPLOS*, pages 457–471, 2016.

[156] Matt Mahoney. Large Text Compression Benchmark. URL http://mattmahoney.net/dc/text.html.

[157] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius,

Xi Wang, and Amin Vahdat. Snap: a microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 399–413, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359657. URL https://doi.org/10.1145/3341301.3359657.

[158] Hasan Al Maruf and Mosharaf Chowdhury. Effectively prefetching remote memory with Leap. In *USENIX ATC*, pages 843–857, 2020.

[159] John C. McCullough, John Dunagan, Alec Wolman, and Alex C. Snoeren. Stout: An adaptive interface to scalable cloud storage. In *USENIX ATC*, 2010.

[160] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Trans. Comput. Syst.*, 2(3):181–197, 1984.

[161] Mellanox. NVMe over fabrics. http://community.mellanox.com/s/article/what-is-nvme-over-fabrics-x.

[162] Mellanox. RDMA aware programming manual (rev. 1.7). https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.

[163] Memcached. A distributed memory object caching system. http://memcached.org, 2022.

[164] Microsoft Azure. Azure spot virtual machines. https://azure.microsoft.com/en-us/pricing/spot, 2022.

[165] Ronald G Minnich and David J Farber. The mether system: Distributed shared memory for sunos 4.0. *Technical Reports (CIS)*, page 332, 1993.

[166] Mistral AI. Mixtral-8x22b: Cheaper, better, faster, stronger. https://mistral.ai/news/mixtral-8x22b/, 2024.

[167] Sparsh Mittal. A survey of recent prefetching techniques for processor caches. *ACM Comput. Surv.*, 49(2), 2016.

[168] MongoDB. https://www.mongodb.com/, 2022.

[169] MongoDB. Wiredtiger storage engine. https://www.mongodb.com/docs/manual/core/wiredtiger/, 2023.

[170] Moonshot AI. AI Assistant with Memory. https://www.perplexity.ai/, 2024.

[171] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, Carlsbad, CA, October 2018. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/moritz.

[172] Mosaic AI Research. Llm inference performance engineering: Best practices. https://www.databricks.com/blog/llm-inference-performance-engineering-best-practices, 2024.

[173] Yogesh Mundada, Anirudh Ramachandran, and Nick Feamster. Silverline: Data and network isolation for cloud services. In *HotCloud*, 2011.

[174] Avanika Narayan, Ines Chami, Laurel Orr, Simran Arora, and Christopher Ré. Can foundation models wrangle your data?, 2022. URL https://arxiv.org/abs/2205.09911.

[175] Shashi Narayan, Shay B. Cohen, and Mirella Lapata. Don't give me the details, just the summary! topic-aware convolutional neural networks for extreme summarization. *ArXiv*, abs/1808.08745, 2018. URL https://api.semanticscholar.org/CorpusID:215768182.

[176] Usama Naseer and Theophilus A. Benson. Configanator: A data-driven approach to improving CDN performance. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1135–1158, Renton, WA, April 2022. USENIX Association. ISBN 978-1-939133-27-4. URL https://www.usenix.org/conference/nsdi22/presentation/naseer.

[177] Christian Navasca, Cheng Cai, Khanh Nguyen, Brian Demsky, Shan Lu, Miryung Kim, and Guoqing Harry Xu. Gerenuk: Thin computation over big native data using speculative program transformation. In *SOSP*, pages 538–553, 2019.

[178] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. {Latency-Tolerant} software distributed shared memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 291–305, 2015.

[179] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *USENIX ATC*, pages 291–305, 2015.

[180] Neo4j. Neo4j graph data platform. https://neo4j.com, 2021.

[181] Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, Jianfei Hu, and Guoqing Xu. Facade: A compiler and runtime for (almost) object-bounded big data applications. In *ASPLOS*, pages 675–690, 2015.

[182] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *OSDI*, pages 349–365, 2016.

[183] Khanh Nguyen, Lu Fang, Christian Navasca, Guoqing Xu, Brian Demsky, and Shan Lu. Skyway: Connecting managed heaps in distributed big data systems. In *ASPLOS*, pages 56–69, 2018.

[184] Nvidia. Virtual gpu (vgpu) | nvidia. https://www.nvidia.com/en-us/data-center/virtual-solutions/.

[185] NVIDIA. Fastertransformer: Transformer related optimization, including bert, gpt. https://github.com/NVIDIA/FasterTransformer, 2024.

[186] NVIDIA. Tensorrt-llm: A tensorrt toolbox for optimized large language model inference. https://github.com/NVIDIA/TensorRT-LLM, 2024.

[187] NVIDIA. Nvidia triton inference server. https://developer.nvidia.com/triton-inference-server, 2024.

[188] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. Cuda, release: 10.2.89, 2020. URL https://developer.nvidia.com/cuda-toolkit.

[189] OpenAI. Chatgpt: Conversational language model. https://chat.openai.com, 2023.

[190] OpenAI. Gpt-4 technical report, 2023.

[191] OpenAI. Introducing the gpt store. https://openai.com/index/introducing-the-gpt-store/, 2024.

[192] OpenAI. Batch api. https://platform.openai.com/docs/guides/batch/batch-api, 2024.

[193] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A fast, extensible toolkit for sequence modeling, 2019.

[194] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *NSDI*, pages 361–378, 2019.

[195] A.K. Parekh and R.G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, 1993. doi: 10.1109/90.234856.

[196] SeongJae Park. Using damon for proactive reclaim. https://lwn.net/Articles/863753/.

[197] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[198] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting, 2024.

[199] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *SOSP*, pages 79–95, 1995.

[200] Leeor Peled, Shie Mannor, Uri Weiser, and Yoav Etsion. Semantic locality and context-based prefetching using reinforcement learning. In *ISCA*, pages 285–297, 2015.

[201] Perlexity AI. Perplexity is a free ai search engine. https://www.perplexity.ai/, 2024.

[202] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *OSDI*, pages 1–16, 2014.

[203] Lucian Popa, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. Faircloud: Sharing the network in cloud computing. In *SIGCOMM*, pages 187–198, 2012.

[204] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 325–341, New York, NY, USA,

2017. Association for Computing Machinery. ISBN 9781450350853. doi: 10.1145/3132747.3132780. URL https://doi.org/10.1145/3132747.3132780.

[205] Proxet. Llm has a performance problem inherent to its architecture: Latency. https://www.proxet.com/blog/llm-has-a-performance-problem-inherent-to-its-architecture-latency, 2023.

[206] Qifan Pu, Haoyuan Li, Matei Zaharia, Ali Ghodsi, and Ion Stoica. FairRide: Near-Optimal, fair cache sharing. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 393–406, Santa Clara, CA, March 2016. USENIX Association. URL https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/pu.

[207] Yifan Qiao, Chenxi Wang, Zhenyuan Ruan, Adam Beley, Yiying Zhang, Miryung Kim, and Guoqing Harry Xu. Hermit: Low-latency, high-throughput, and transparent remote memory via feedback-directed asynchrony. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, April 2023. URL https://www.usenix.org/conference/nsdi23/presentation/qiao.

[208] Yifan Qiao, Zhenyuan Ruan, Haoran Ma, Adam Belay, Miryung Kim, and Harry Xu. Harvesting idle memory for application-managed soft state with midas. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1247–1265, Santa Clara, CA, April 2024. USENIX Association. ISBN 978-1-939133-39-7. URL https://www.usenix.org/conference/nsdi24/presentation/qiao.

[209] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: A kvcache-centric disaggregated architecture for llm serving, 2024. URL https://arxiv.org/abs/2407.00079.

[210] Haonan Qiu, Xiaoliang Wang, Tianchen Jin, Zhuzhong Qian, Baoliu Ye, Bin Tang,

Wenzhong Li, and Sanglu Lu. Toward effective and fair RDMA resource sharing. In *APNet*, pages 8–14, 2018.

[211] Rodric M. Rabbah, Hariharan Sandanagobalane, Mongkol Ekpanyapong, and Weng-Fai Wong. Compiler orchestrated prefetching via speculation and predication. In *ASPLOS*, pages 189–198, 2004.

[212] Benjamin Reidys, Jinghan Sun, Anirudh Badam, Shadi Noghabi, and Jian Huang. BlockFlex: Enabling storage harvesting with Software-Defined flash in modern cloud platforms. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 17–33, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/reidys.

[213] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models, 2021.

[214] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015. URL https://networkrepository.com.

[215] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024.

[216] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-performance, application-integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 315–332. USENIX

Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/ruan.

[217] Zhenyuan Ruan, Shihang Li, Kaiyan Fan, Marcos K. Aguilera, Adam Belay, Seo Jin Park, and Malte Schwarzkopf. Unleashing true utility computing with quicksand. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HOTOS '23, page 196–205, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701955. doi: 10.1145/3593856.3595893. URL https://doi.org/10.1145/3593856.3595893.

[218] Zhenyuan Ruan, Seo Jin Park, Marcos Aguilera, Adam Belay, and Malte Schwarzkopf. Nu: Achieving microsecond-scale resource fungibility with logical processes. In *NSDI*, 2023.

[219] Stephen M. Rumble. Infiniband verbs performance. https://ramcloud.atlassian.net/wiki/display/RAM/Infiniband+Verbs+Performance, 2010.

[220] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. Log-structured memory for DRAM-based storage. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 1–16, Santa Clara, CA, February 2014. USENIX Association. ISBN ISBN 978-1-931971-08-9. URL https://www.usenix.org/conference/fast14/technical-sessions/presentation/rumble.

[221] Stuart J Russell. *Artificial intelligence a modern approach*. Pearson Education, Inc., 2010.

[222] Alon Shalita, Brian Karrer, Igor Kabiljo, Arun Sharma, Alessandro Presta, Aaron Adcock, Herald Kllapi, and Michael Stumm. Social hash: an assignment framework for optimizing distributed systems operations on social networks. In *13th* {*USENIX*}

Symposium on Networked Systems Design and Implementation ({NSDI} 16), pages 455–468, 2016.

[223] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 323–337, 2017.

[224] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *OSDI*, pages 69–87, 2018.

[225] Prateek Sharma, Ahmed Ali-Eldin, and Prashant Shenoy. Resource deflation: A new approach for transient resource reclamation. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362818. doi: 10.1145/3302424.3303945. URL https://doi.org/10.1145/3302424.3303945.

[226] Noam Shazeer. Fast transformer decoding: One write-head is all you need, 2019.

[227] Dian Shen, Junzhou Luo, Fang Dong, Xiaolin Guo, Kai Wang, and John C. S. Lui. Distributed and optimal rdma resource scheduling in shared data center networks. In *INFOCOM*, pages 606–615, 2020.

[228] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: high-throughput generative inference of large language models with a single gpu. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023.

[229] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, Joseph E. Gonzalez, and Ion Stoica. S-lora: Serving thousands of concurrent lora adapters, 2024. URL https://arxiv.org/abs/2311.03285.

[230] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E. Gonzalez, and Ion Stoica. Fairness in serving large language models. In *18th USENIX*

Symposium on Operating Systems Design and Implementation (OSDI 24), pages 965–988, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/sheng.

[231] Timothy Sherwood, Suleyman Sair, and Brad Calder. Predictor-directed stream buffers. In *MICRO*, pages 42–53, 2000.

[232] Alan Shieh, Srikanth Kandula, Albert Greenberg, and Changhoon Kim. Seawall: Performance isolation for cloud datacenter networks. In *HotCloud*, 2010.

[233] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using gpu model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[234] David Shue, Michael J. Freedman, and Anees Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *OSDI*, pages 349–362, 2012.

[235] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. Strom: Smart remote memory. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450368827. doi: 10.1145/3342195.3387519. URL https://doi.org/10.1145/3342195.3387519.

[236] Akshitha Sriraman and Thomas F. Wenisch. µsuite: A benchmark suite for microservices. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–12, 2018. doi: 10.1109/IISWC.2018.8573515.

[237] Robert Stets, Sandhya Dwarkadas, Nikolaos Hardavellas, Galen Hunt, Leonidas Kontothanassis, Srinivasan Parthasarathy, and Michael Scott. Cashmere-2l: Software coherent shared memory on a clustered remote-write network. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 170–183, 1997.

[238] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic scheduling for large language model serving, 2024.

[239] Zhenheng Tang, Yuxin Wang, Qiang Wang, and Xiaowen Chu. The impact of gpu dvfs on the energy and performance of deep learning: an empirical study. In *Proceedings of the Tenth ACM International Conference on Future Energy Systems*, e-Energy '19, page 315–325, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450366717. doi: 10.1145/3307772.3328315. URL https://doi.org/10.1145/3307772.3328315.

[240] Konstantin Taranov, Salvatore Di Girolamo, and Torsten Hoefler. Corm: Compactable remote memory over rdma. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1811–1824, 2021.

[241] Kiran Tati and Geoffrey M. Voelker. Shortcuts: Using soft state to improve dht routing. In Chi-Hung Chi, Maarten van Steen, and Craig Wills, editors, *Web Content Caching and Distribution*, page 44–62, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-30471-5.

[242] The Claude Team. Introducing the next generation of claude. https://www.anthropic.com/news/claude-3-family, 2024.

[243] The PyTorch Foundation. Torchserve is a performant, flexible, and easy to use tool for serving pytorch models in production. https://pytorch.org/serve/, 2024.

[244] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. IOFlow: A software-defined storage architecture. In *SOSP*, pages 182–196, 2013.

[245] Tien-Fu Chen and Jean-Loup Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44(5):609–623, 1995.

[246] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: The next generation. In *EuroSys*, 2020.

[247] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

[248] Shin-Yeh Tsai and Yiying Zhang. LITE kernel RDMA support for datacenter applications. In *SOSP*, pages 306–324, 2017.

[249] Twitter Inc. Processing billions of events in real time at twitter. https://blog.twitter.com/engineering/en_us/topics/infrastructure/2021/processing-billions-of-events-in-real-time-at-twitter-, 2021.

[250] S. P. Vander Wiel and D. J. Lilja. When caches aren't enough: data prefetching techniques. *Computer*, 30(7):23–30, 1997.

[251] S. P. Vander Wiel and D. J. Lilja. A compiler-assisted data prefetch controller. In *Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 372–377, 1999.

[252] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[253] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[254] Lluís Vilanova, Lina Maudlej, Shai Bergman, Till Miemietz, Matthias Hille, Nils Asmussen, Michael Roitzsch, Hermann Härtig, and Mark Silberstein. Slashing the

disaggregation tax in heterogeneous data centers with fractos. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 352–367, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391627. doi: 10.1145/3492321.3519569. URL https://doi.org/10.1145/3492321.3519569.

[255] Geoffrey M. Voelker, Eric J. Anderson, Tracy Kimbrel, Michael J. Feeley, Jeffrey S. Chase, Anna R. Karlin, and Henry M. Levy. Implementing cooperative prefetching and caching in a globally-managed memory system. In *SIGMETRICS*, pages 33–43, 1998.

[256] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building an elastic query engine on disaggregated storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 449–462, Santa Clara, CA, February 2020. USENIX Association. ISBN 978-1-939133-13-7. URL https://www.usenix.org/conference/nsdi20/presentation/vuppalapati.

[257] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. Argon: Performance insulation for shared storage servers. In *FAST*, 2007.

[258] 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 *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 261–280. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/wang.

[259] 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 *OSDI*, 2022.

[260] Chenxi Wang, Yifan Qiao, Haoran Ma, Shi Liu, Yiying Zhang, Wenguang Chen, Ravi

Netravali, Miryung Kim, and Guoqing Harry Xu. Canvas: Isolated and adaptive swapping for multi-applications on remote memory. In *NSDI*, 2023.

[261] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. Concordia: Distributed shared memory with {In-Network} cache coherence. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 277–292, 2021.

[262] Xiaodong Wang and José F. Martínez. ReBudget: Trading off efficiency vs. fairness in market-based multicore resource allocation via runtime budget reassignment. In *ASPLOS*, pages 19–32, 2016.

[263] Yuxin Wang, Yuhan Chen, Zeyu Li, Zhenheng Tang, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. Towards efficient and reliable llm serving: A real-world workload study, 2024.

[264] Qizhen Weng, Lingyun Yang, Yinghao Yu, Wei Wang, Xiaochuan Tang, Guodong Yang, and Liping Zhang. Beware of fragmentation: Scheduling GPU-Sharing workloads with fragmentation gradient descent. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 995–1008, Boston, MA, July 2023. USENIX Association. ISBN 978-1-939133-35-9. URL https://www.usenix.org/conference/atc23/presentation/weng.

[265] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. Transparent GPU sharing in container clouds for deep learning workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 69–85, Boston, MA, April 2023. USENIX Association. ISBN 978-1-939133-33-5. URL https://www.usenix.org/conference/nsdi23/presentation/wu.

[266] Bingyang Wu, Ruidong Zhu, Zili Zhang, Peng Sun, Xuanzhe Liu, and Xin Jin. dLoRA: Dynamically orchestrating requests and adapters for LoRA LLM serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*,

pages 911–927, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/wu-bingyang.

[267] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks, 2024. URL https://arxiv.org/abs/2309.17453.

[268] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, Carlsbad, CA, October 2018. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/xiao.

[269] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. AntMan: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 533–548. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/xiao.

[270] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-Flux: Precise online qos management for increased utilization in warehouse scale computers. In *ISCA*, pages 607–618, 2013.

[271] Suli Yang, Tyler Harter, Nishant Agrawal, Salini Selvaraj Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Rini T. Kaushik, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Split-level i/o scheduling. In *SOSP*, pages 474–489, 2015.

[272] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th*

USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), pages 521–538, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/yu.

[273] Peifeng Yu and Mosharaf Chowdhury. Salus: Fine-grained gpu sharing primitives for deep learning applications, 2019.

[274] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *HotCloud*, HotCloud, page 10, Berkeley, CA, USA, 2010.

[275] Matei Zaharia, Omar Khattab, Lingjiao Chen, Jared Quincy Davis, Heather Miller, Chris Potts, James Zou, Michael Carbin, Jonathan Frankle, Naveen Rao, and Ali Ghodsi. The shift from models to compound ai systems. https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/, 2024.

[276] Haoran Zhang, Konstantinos Kallas, Spyros Pavlatos, Rajeev Alur, Sebastian Angel, and Vincent Liu. MuCache: A general framework for caching in microservice graphs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 221–238, Santa Clara, CA, April 2024. USENIX Association. ISBN 978-1-939133-39-7. URL https://www.usenix.org/conference/nsdi24/presentation/zhang-haoran.

[277] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. SHEPHERD: Serving DNNs in the wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 787–808, Boston, MA, April 2023. USENIX Association. ISBN 978-1-939133-33-5. URL https://www.usenix.org/conference/nsdi23/presentation/zhang-hong.

[278] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar,

Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *SOSP*, pages 195–211, 2021.

[279] Lixia Zhang. A new architecture for packet switching network protocols. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE, 1989.

[280] Tianjun Zhang, Shishir G. Patil, Naman Jain, Sheng Shen, Matei Zaharia, Ion Stoica, and Joseph E. Gonzalez. Raft: Adapting language model to domain specific rag, 2024.

[281] Wei Zhang, Sundaresan Rajasekaran, Shaohua Duan, Timothy Wood, and Mingfa Zhuy. Minimizing interference and maximizing progress for hadoop virtual machines. *SIGMETRICS Perform. Eval. Rev.*, 42(4):62–71, 2015.

[282] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. CPI^2: CPU performance isolation for shared compute clusters. In *EuroSys*, pages 379–391, 2013.

[283] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 724–739, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450387095. doi: 10.1145/3477132.3483580. URL https://doi.org/10.1145/3477132.3483580.

[284] Yiwen Zhang, Yue Tan, Brent E. Stephens, and Mosharaf Chowdhury. RDMA performance isolation with justitia. In *NSDI*, 2022.

[285] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving, 2024.

[286] D. F. Zucker, R. B. Lee, and M. J. Flynn. Hardware and software cache prefetching techniques for MPEG benchmarks. *IEEE Transactions on Circuits and Systems for Video Technology*, 10(5):782–796, 2000.