Practical Memory Disaggregation

by

Md Hasan Al Maruf

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2023

Doctoral Committee:

Associate Professor Mosharaf Chowdhury, Chair
Assistant professor Asaf Cidon, Columbia University
Assistant Professor Baris Kasikci
Professor Lei Ying

Md Hasan Al Maruf
hasanal@umich.edu
ORCID iD: 0000-0003-0089-9370

to my family

for their love, support, and encouragement

ACKNOWLEDGMENTS

The completion of my PhD journey is a testament to the support and encouragement of an incredible community. To all those who have played a role, whether large or small, in shaping this endeavor, I offer my heartfelt gratitude.

I am highly grateful to my advisor, Mosharaf Chowdhury, for his continuous support, patience, and guidance throughout my graduate studies. For me, Mosharaf has been an ideal mentor – he believed in my potential and gave me enough freedom to explore on my own. At the same time, his thoughtful insights and honest feedback helped me be an improved researcher and a person. Mosharaf's insistence on doing impactful research motivated me to work on problems that matter in the real world. I enjoyed my time working with him.

This dissertation is the culmination of many successful collaborations. I am thankful to Youngmoon Lee and Asaf Cidon for their ideas that surely strengthen Chapter 3. Chapter 4 is a joint work with Asaf Cidon and Carl Waldspurger. Thanks to Yuhong Zhong and Hongyi Wang for their help in running experiments and wrapping up the project. I am indebted to Meta for providing me early access to the CXL-enabled system to continue my work on Chapter 5. I enjoyed my time at Meta as a visiting researcher. I am grateful to all CEA team members, especially Abhishek Dhanotia, Hao Wang, Niket Agarwal, and Pallab Bhattacharya, for their valuable feedback and guidance.

I want to extend my sincere gratitude to all the current and past members of Symbiotic Lab – especially Juncheng Gu, Youngmoon Lee, Fan Lai, Yiwen Zhang, Jie You, Peifeng Yu, Jiachen Liu, Jae-Won Chung, and Insu Jang. I enjoyed every discussion and brainstorm sessions we had in our lab, and all the fun we have had in the last several years. In particular, I am grateful to Juncheng and Youngmoon for their kind help and guidance during the very first year of my PhD. I will always miss the hallway chats and laughters with Fan and Yiwen. Besides Symbiotic Lab members, many friends in Ann Arbor enriched my life outside work. Among them, special thanks to Kamruzzaman Khan and Mushfequr Rahman for countless memories and all the meaningless chats over the coffee. I also thank my numerous friends and well-wishers all over the world.

I want to express my deepest gratitude to my parents and siblings for their love and support. My mother, Sayeda Parveen, is always a source of inspiration to me. None of my achievements would have been possible without her endless affection, infinite patience, and countless sacrifices to ensure a better future for me. Throughout my life, I admire my father, Abu Siddique, for his

"take it easy" attitude toward difficult situations in life. I am grateful to my eldest sister Salmin Sultana and brother-in-law Iftekhar Alam for their support. Whenever I faced hard times in my USA life, I somehow ended up with their family and got charged back. Their child, Srija, is always a bundle of joy and close to my heart. I am thankful to my younger sister Sabia Sultana for all her carings and generosities. KoToRo helped me survive the winter blizzard of Michigan through his warm furrs, sharp bites, and endless purrs.

I will always cherish the last winter of my PhD life at Ann Arbor. It was a short one but pretty intense. Although I used to complain about the bleak and harsh Michigan winter, this time, I loved the elegance of snowflakes. I am happy that eventually, I could feel the silent symphony of snow falls and enjoy the tranquility it brought.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE

LIST OF TABLES

TABLE

LIST OF ACRONYMS

FIFO First In, First Out

CXL Compute Express Link

DIMM Dual In-line Memory Module

DRAM Dynamic Random Access Memory

GFAM Global Fabric Attached Memory

HDD Hard Disk Drive

IPT Idle Page Tracking

LPDDR Low-Power Double Data Rate

NIC Network Interface Card

NUMA Non-Uniform Memory Access

NUMA Non-Volatile Memory

OS Operating System

PCIe Peripheral Component Interconnect Express

PEBS Precise Event-Based Sampling

QoS Quality of Service

RDMA Remote Direct Memory Access

SLO Service Level Objective

SSD Solid State Drive

TCO Total Cost of Ownership

VFS Virtual File System

VM Virtual Memory

VMM Virtual Memory Manager

ABSTRACT

In today's datacenters, compute and memory resources are tightly-coupled. This causes fleet-wide resource underutilization and increases the total cost of ownership (TCO) for large-scale datacenters. Modern datacenters are embracing a paradigm shift towards disaggregation, where each resource type is decoupled and connected through a network fabric. As memory is the prime resource for high-performance services, it has become an attractive target for disaggregation. Disaggregating memory from compute enables flexibility to scale them independently and better resource utilization. As memory consumes 30-40% of the total rack power and operating cost, proper utilization of stranded resources through disaggregation can save billions of dollars in TCO.

With the advent of ultra-fast networks and coherent interfaces like CXL, disaggregation has become popular over the last few years. There are, however, many open challenges for its practical adoption, including the latency gap between local and remote memory access, resilience, deployability in existing infrastructure, adaptation of heterogeneity in cluster resources, and isolation while maintaining the quality of service. To make memory disaggregation widely adoptable, besides hardware support, we need to provide performant software stacks considering all these challenges so that such systems do not degrade application performance beyond a noticeable margin.

This dissertation proposes a comprehensive solution to address the host-level, network-level, and end-to-end aspects of practical memory disaggregation. Existing memory disaggregation solutions usually use data path components designed for slow disks. As a result, applications experience remote memory access latency significantly higher than that of the underlying low-latency network. To bridge the still-sizeable latency gap between local vs. remote memory access, we design **Leap** – a prefetching solution for remote memory accesses. At its core, Leap employs an online, majority-based prefetching algorithm, which increases the page cache hit rate. Next, comes the challenge of providing resilience. Relying on memory across multiple machines in a disaggregated cluster makes applications susceptible to a wide variety of uncertainties, such as independent and correlated failures of remote machines, evictions from and corruptions of remote memory, network partitions, etc. Applications also suffer from stragglers or late-arriving remote responses because of the latency variabilities in a large network due to congestion and background traffic. **Hydra** addresses these issues by enabling a low-latency, low-overhead, and highly available erasure-coded resilient remote memory datapath at single-digit μs tail latency.

For widespread deployability, besides private clouds, we consider public clouds where prevail a set of unique challenges for resource disaggregation across different tenants, including resource harvesting, isolation, and matching. We design **Memtrade** to enable memory disaggregation on public clouds even in the absence of the latest networking hardware and protocols (e.g., RDMA, CXL). Memtrade allows producer virtual machines (VMs) to lease both their unallocated memory and allocated-but-idle application memory to remote consumer VMs for a limited period of time.

Emerging coherent interfaces like CXL reduce the remote memory access latency to a few hundred nanoseconds. It enables main memory expansion where different memory technologies with varied characteristics can co-exist. Without efficient memory management, however, such heterogeneous tiered-memory systems can significantly degrade performance. We propose a novel OS-level page placement mechanism, **TPP**, for tiered-memory systems. TPP employs a lightweight mechanism to identify and place hot/cold pages to appropriate memory tiers.

Altogether, this dissertation presents how to enable practical memory disaggregation for next-generation datacenters through performant, resilient, and easily deployable software stacks.

CHAPTER 1

Introduction

Modern datacenter applications are becoming more and more memory intensive to meet their unyielding demand for low-latency and high throughput services. Emerging applications in different service domains like graph processing [217, 162], stream processing [238, 116, 315], interactive analysis [316, 64], SQL processing [64, 102, 195, 84], machine learning [160, 196, 228], deep learning [74, 122, 174], etc. require high memory demand to avoid expensive disk accesses which drastically degrades an application's performance by orders of magnitude. For example, if one runs an iterative machine learning algorithm such as logistic regression on Apache Spark, in-memory configuration provides $100\times$ better performance over a disk-based Apache Hadoop setup [14].

With the increase in data volume, total service queries, number of end-users, etc. factors, the overall memory demand of these applications can increase over time. Because of a server's physical memory capacity limit, if the memory requirement increases drastically, it becomes impossible to serve the whole working set of an application by a single server. At that point, memory-intensive applications experience disproportionate performance loss. Prior research [165] showed that, for a suite of memory-intensive applications, if the available memory of a host server cannot serve half of the application's working set, the performance loss can vary by $8\times$ to $25\times$ (Figure 1.1).

Application developers often sidestep such disasters by over-allocating memory, but pervasive over-allocation inevitably leads to datacenter-scale memory underutilization. Indeed, memory utilization at many hyperscalers hovers around 40%–60% [263, 4, 165, 202]. Service providers running on public clouds, such as Snowflake, report 70%–80% underutilized memory on average [292]. Moreover, in many real-world deployments, workloads rarely use all of their allocated memory all of the time. Often, an application allocates a large amount of memory but accesses it infrequently during its remaining life cycle. For example, in Google's datacenters, up to 61% of allocated memory remains idle [202]. Since DRAM is a significant driver of infrastructure cost and power consumption [146, 34, 147], excessive underutilization leads to high capital and operating expenditures, as well as wasted energy (and carbon emissions).

At the same time, increasing the effective memory capacity and bandwidth of each server to accommodate ever-larger working sets is challenging as well. More importantly, memory capacity

comes at power-of-two granularity which limits finer grain memory capacity sizing. Even numbers of memory controllers need to be presented with memory DIMMs installed in pairs. All the memory channels needs to run at the same speed and bandwidth. At the same time, as DRAM continues to densify with newer generations, to increase the memory bandwidth of a platform, one needs to increase a server's memory capacity. For example, to increase memory bandwidth by $3.6\times$ in their datacenters, Meta had to increase capacity by $16\times$ [225]. To provide sufficient memory capacity and/or bandwidth to memory-intensive applications, computing and networking resources become stranded in traditional server platforms, which eventually causes fleet-wide resource underutilization and increases TCO.

1.1 Memory Disaggregation

Memory disaggregation addresses memory-related rightsizing problems at both software and hardware levels. Applications are able to allocate memory as they need without being constrained by server boundaries. Servers are not forced to add more computing and networking resources when they only need additional memory capacity or bandwidth.

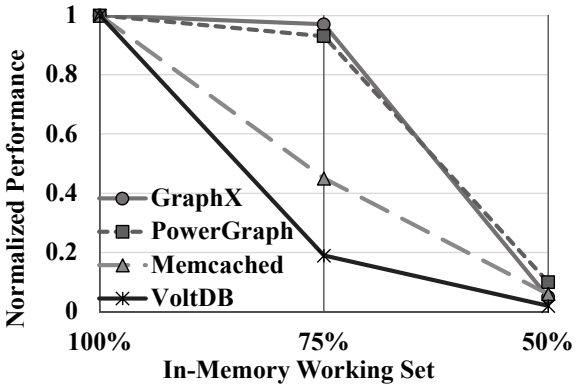


Figure 1.1: Memory-intensive applications suffer disproportionate loss when working sets do not fit

Simply put, memory disaggregation exposes memory capacity available in remote locations as a pool of memory and shares it across multiple servers over the network. It decouples the available compute and memory resources, enabling independent resource allocation in the cluster. A server's local and remote memory together constitute its total physical memory. An application's locality of memory reference allows the server to exploit its fast local memory to maintain high performance, while remote memory provides expanded capacity with an increased access latency that is still orders-of-magnitude faster than accessing persistent storage (e.g., HDD, SSD). The OS and/or application runtime provides the necessary abstractions to expose all the available memory in the cluster, hiding the complexity of setting up and accessing remote memory (e.g., connection setup, memory access semantics, network packet scheduling, etc.) while providing resilience, isolation, security, etc. guarantees.

By exposing all unused memory across all the servers as a memory pool to all memory-

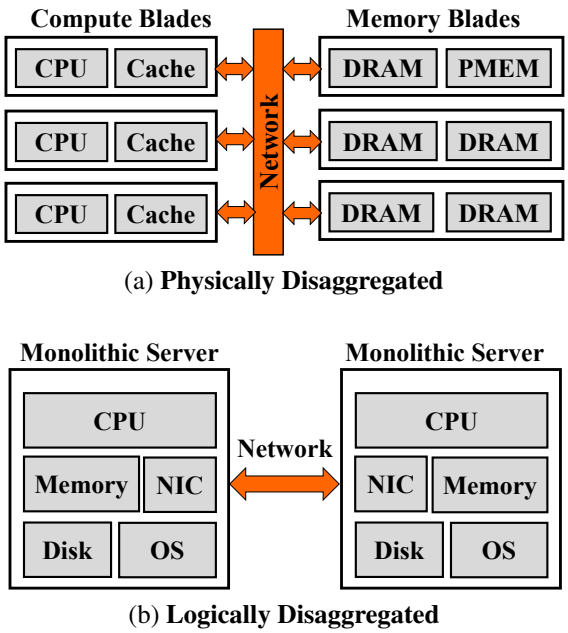


Figure 1.2: Physical vs. logical memory disaggregation architectures.

intensive applications, memory disaggregation can improve both application-level performance and overall memory utilization. Multiple hardware vendors and hyperscalers have projected [66, 73, 208, 225] up to 25% TCO savings without affecting application performance via (rack-scale) memory disaggregation. While the idea of leveraging remote machines' memory is decades old [120, 145, 151, 210, 223, 241], only during the past few years, the latency and bandwidth gaps between memory and communication technologies have come close enough to make it practical. The first disaggregated memory[1] solutions [93, 157, 165, 258] leveraged RDMA over InfiniBand or Ethernet, but they are an order-of-magnitude slower than local memory. More recently, with the rise of cache-coherent Compute Express Link (CXL) [27] interconnects and hardware protocols, the latency and performance gap is decreasing even more. We are at the cusp of taking a leap toward next-generation software-hardware co-designed disaggregated memory systems.

1.1.1 Memory Disaggregation Architectures

Memory disaggregation systems have two primary cluster memory architectures.

Physical Disaggregation. In a physically-disaggregated architecture, compute and memory nodes are detached from each other where a cluster of compute blades are connected to one or more memory blades through network (e.g., PCIe bridge) [214] (Figure 1.2a). A memory node

[1]Remote memory and far memory are often used interchangeably with the term disaggregated memory.

can be a traditional monolithic server with low compute resource and large memory capacity, or it can be network-attached DRAM. For better performance, the compute nodes are usually equipped with a small amount of memory for caching purposes.

Logical Disaggregation. In a logically-disaggregated architecture, traditional monolithic servers hosting both compute and memory resources are connected to each other through the network (e.g., Infiniband, RoCEv2) (Figure 1.2b). This is a popular approach for building a disaggregated memory system because one does not need to change existing hardware architecture; simply incorporating appropriate software to provide a remote memory interface is sufficient. In such a setup, usually, each of the monolithic servers has their own OS. In some cases, the OS itself can be disaggregated across multiple hosts [275]. Memory local to a host is usually prioritized for running local jobs. Unutilized memory on remote machines can be pooled and exposed to the cluster as remote [165, 224, 207, 226, 275, 267, 111].

Hybrid Approach. Cache-coherent interconnects like CXL provides the opportunity to build a composable heterogeneous server memory systems that combine logical and physical disaggregation approaches. Multiple monolithic servers, compute devices, memory nodes, or network specialized devices can be connected through fabric or switches where software stacks can provide the cache-line granular or traditional virtual memory-based disaggregated memory abstraction.

1.1.2 Abstractions and Interfaces

Interfaces to access disaggregated memory can either be transparent to the application or need minor to complete re-write of applications (Table 1.1). The former has broader applicability, while the latter might have better performance.

Application-Transparent Interface. Access to remote disaggregated memory without significant application rewrites typically relies on two primary mechanisms: disaggregated Virtual File System (VFS) [92], that exposes remote memory as files and disaggregated Virtual Memory Manager (VMM) for remote memory paging [224, 165, 207, 275]. In both cases, data is communicated in small chunks or pages (typically, 4KB). In case of remote memory as files, pages go through the file system before they are written to/read from the remote memory. For remote memory paging and distributed OS, page faults cause the VMM to write pages to and read them from the remote memory. Remote memory paging is more suitable for traditional applications because it does not require software or hardware modifications.

Abstraction	System	Hardware Transparent	OS Transparent	Application Transparent
Virtual Memory Management (VMM)	Global Memory [149]	Yes	No	Yes
	Memory Blade [214]	No	No	Yes
	Infiniswap [165]	Yes	Yes	Yes
	Leap [224]	Yes	No	Yes
	LegoOS [275]	Yes	No	Yes
	zSwap [202]	Yes	No	Yes
	Kona [111]	Yes	No	Yes
	Fastswap [98]	Yes	No	Yes
	Hydra [207]	Yes	Yes	Yes
Virtual File System (VFS)	Memory Pager [223]	Yes	Yes	No
	Remote Regions [92]	Yes	Yes	No
Custom API	FaRM [142]	Yes	Yes	No
	FaSST [180]	Yes	Yes	No
	LITE [288]	Yes	Yes	No
	Memtrade [226]	Yes	Yes	No
Programming Runtime	AIFM [267]	Yes	Yes	No
	Semeru [297]	Yes	Yes	No
	Nu [266]	Yes	Yes	No

Table 1.1: Selected memory disaggregation proposals.

Non-Transparent Interface. Another approach is to directly expose remote memory through custom API (KV-store, remote memory-aware library or system calls) and modify the applications incorporating these specific APIs [267, 297, 180, 142, 266, 226]. All the memory (de)allocation, transactions, synchronizations, etc. operations are handled by the underlying implementations of these APIs. Performance optimizations like caching, local-vs-remote data placement, prefetching, etc. are often the responsibility of the application.

1.2 Challenges in Practical Memory Disaggregation

Simply relying on fast networks or interconnects is not sufficient to make memory disaggregation practical. A comprehensive solution must address challenges in multiple dimensions:

- **High Performance.** A disaggregated memory system involves the network in its remote memory path, which is at least an order-of-magnitude slower than memory channels attached to CPU and DRAM (80–140 nanoseconds vs. microseconds; see Figure 1.3). Hardware-induced remote memory latency is significant and impacts application performance [165, 224, 225]. Depending on the abstraction, software stacks can also introduce significant overheads. For example, remote memory paging over existing OS VMM can add tens of microseconds latency for a 4KB page [224].

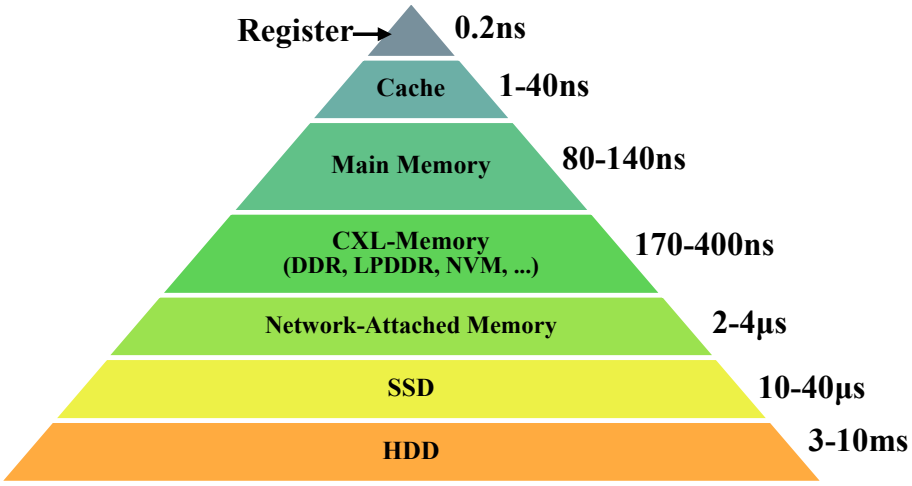


Figure 1.3: Latency characteristics of different memory technologies.

- **Performance Isolation.** When multiple applications with different performance requirements (e.g., latency- vs. bandwidth-sensitive workloads) compete for remote memory, depending on where the applications are running and remote memory location, they may be contending for resources inside the server, on the NIC, in the network on the hardware side, and variety of resources in the application runtimes and OSes. This is further exacerbated by the presence of multiple tiers of memory with different latency-bandwidth characteristics.

- **Resilience to Expanded Failure Domains.** Applications relying on remote memory become susceptible to new failure scenarios such as independent and correlated failures of remote machines, evictions from and corruptions of remote memory, and network partitions. They also suffer from stragglers or late-arriving remote responses due to network congestion and background traffic [139]. These uncertainties can lead to catastrophic failures and service-level objective (SLO) violations.

- **Memory Heterogeneity.** Memory hierarchy within a server is already heterogeneous (Figure 1.3). Disaggregated memory – both network-attached and emerging CXL-memory [208, 163, 225] – further increases heterogeneity in terms of latency-bandwidth characteristics. In such a heterogeneous setup, simply allocating memory to applications is not enough. Instead, decisions like how much memory to allocate in which tier at what time is critical.

- **Efficiency and Scalability.** Disaggregated memory systems are inherently distributed. As the number of memory servers, the total amount of disaggregated memory, and the number of applications increase, the complexity of finding unallocated remote memory in a large cluster, allocating them to applications without violating application-specific SLOs, and cor-

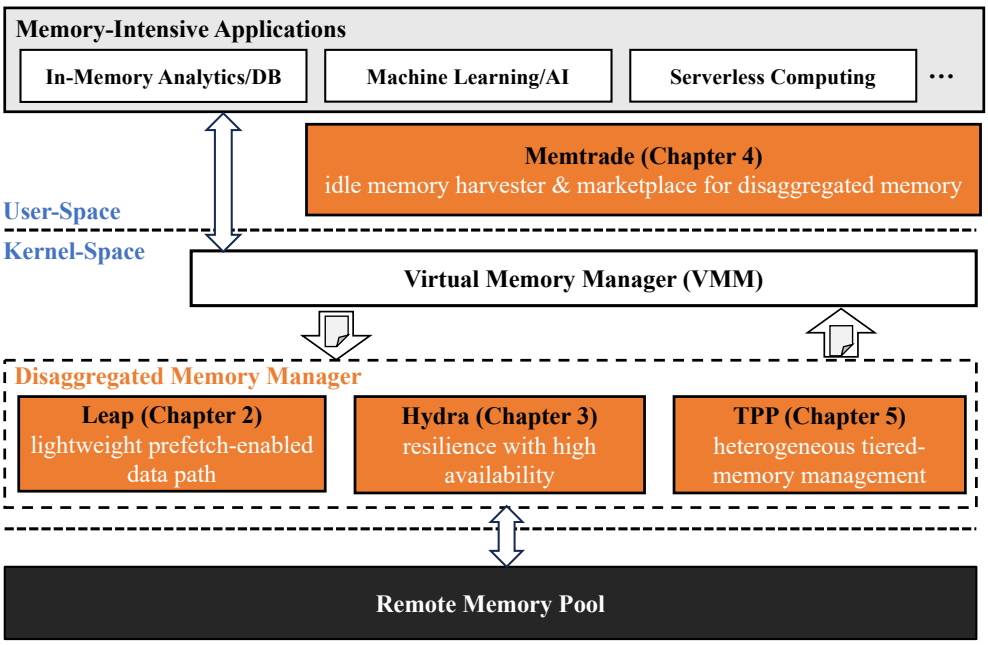


Figure 1.4: Application-transparent software solutions to address different practicality challenges in a logically disaggregated memory architecture.

responding meta-data overhead of memory management increase as well. Finding efficient matching at scale is necessary to achieve high overall utilization.

- **Security.** Although security of disaggregated memory is often sidestepped within the confines of a private datacenter, it is a major challenge for memory disaggregation in public clouds. Since data residing in remote memory may be read by entities without proper access, or corrupted from accidents or malicious behavior, the confidentiality and integrity of remote memory must be protected. Additional concerns include side channel and remote rowhammer attacks over the network [287, 286], distributed coordinated attacks, lack of data confidentiality and integrity and client accountability during CPU bypass operations (e.g., when using RDMA for memory disaggregation).

Thesis Statement. *Practical realization of memory disaggregation can help bend the curve on Moore's law by enabling higher generation of memory technologies on general purpose processors. There are, however, many open challenges including the latency gap between local vs. remote memory access, resilience, deployability, and heterogeneity in hardware infrastructure. This dissertation presents a comprehensive application-transparent software solution for the practical adaptation of memory disaggregation that will scale without losing efficiency down the line.*

We propose a suit of software solutions (Figure 1.4) that augments the VMM of existing OSes by incorporating remote memory awareness in the data path. We fasten the VMM's access to

remote memory through a prefetch-enabled leaner data path (**Leap**). To incorporate resilience, we adopt erasure-coding during remote read/write operations (**Hydra**). For optimum performance, we further enhance the VMM to identify the hot-warm-cold memory regions and put them to appropriate memory tiers across a heterogeneous tiered-memory system (**TPP**). We also expand the capacity of remote memory pool through an efficient idle memory harvesting mechanism and exposing them to the global memory pool (**Memtrade**). For the widespread adaptability, we design this framework so that it can operate even without any specialized hardware and OS modification.

1.3 Contribution

Leap The prime challenge for the practical realization of memory disaggregation is how to bridge the latency gap. Application-transparent paging-based remote memory abstraction incurs high latency overhead because Linux VMM is not optimized for microsecond-scale operations. In this regard, an optimized remote memory data path is imperative. We design Leap that provides a lightweight and efficient data path in the kernel. Leap isolates each application's data path to the disaggregated memory and mitigates latency bottlenecks arising from legacy throughput-optimizing operations. Even with the leanest data path, a reactive page fetching system must suffer microsecond-scale network latency on the critical path. To address this, we complement Leap with a remote memory prefetcher to proactively bring in the correct pages into a local cache to provide sub-microsecond latency (comparable to that of a local page access) on cache hits.

Hydra To tolerate remote failures, memory disaggregation frameworks often relies on local disks, which results in slow failure recovery. Maintaining multiple in-memory replicas may not an option either as it effectively halves the total capacity. We explore erasure coding as a memory-efficient alternative. Specifically, we divide each page into k splits to generate r encoded parity splits and spread the $(k + r)$ splits to $(k + r)$ failure domains – any k out of $(k + r)$ splits would then suffice to decode the original data. However, erasure coding is traditionally applied to large objects [260]. In this regard, we design Hydra whose carefully designed data path can perform online erasure coding within a single-digit microsecond tail latency. Hydra also introduce CodingSets, a new data placement scheme that balance availability and load balancing, while reducing the probability of data loss by an order of magnitude even under large correlated failures.

Memtrade Memory disaggregation frameworks are designed mostly for cooperative private datacenters; memory disaggregation in public clouds faces additional concerns. For the widespread adaptation, we design Memtrade to harvest all the idle memory within virtual machines (VMs) – be it unallocated, or allocated to an application but infrequently utilized, and expose them to a

publicly accessible disaggregated memory marketplace. Memtrade allows producer VMs to lease their idle application memory to remote consumer VMs for a limited period of time while ensuring confidentiality and integrity. It employs a broker to match producers with consumers while satisfying performance constraints. Memtrade enables memory disaggregation on public clouds even without any support from the cloud provider or any special hardware.

TPP Datacenter applications have diverse memory access latency and bandwidth requirements. Sensitivity toward different memory page types can also vary across applications. Understanding and characterizing such behaviors is critical to designing heterogeneous tiered-memory systems. We build **Chameleon**, a lightweight user-space memory access behavior characterization tool that can readily be deployed in production without disrupting running application(s) or modifying the OS kernel. It generates a heat-map of memory usage for different page types and provides insights into an application's expected performance with multiple temperature tiers. Given applications' page characterizations, TPP provides an OS-level transparent page placement mechanism, to efficiently place pages in a tiered-memory system. TPP has three components: **(a)** a lightweight reclamation mechanism to demote colder pages to the slow tier; **(b)** decoupling the allocation and reclamation logic for multi-NUMA systems to maintain a headroom of free pages on the fast tier; and **(c)** a reactive page promotion mechanism that efficiently identifies hot pages trapped in the slow memory tier and promote them to the fast memory tier to improve performance. It also introduces support for page type-aware allocation across the memory tiers.

Our designed software systems together can provide near-memory performance for most memory-intensive applications even when 75% and sometimes more of their working sets reside in remote memory in an application- and hardware-transparent manner, in the presence of failures, load imbalance, and multiple tenants. We believe, our solutions can play a vital role towards the widespread adaptation of memory disaggregation.

1.4 Organization

The remainder of this dissertation is organized as follows. In Chapter 2, we analyze the data path latency overheads for disaggregated memory systems and present Leap to provide sub-microsecond remote memory access. In Chapter 3, we discuss the performance vs efficiency tradeoff while providing resilience and present Hydra – an erasure-coded drop-in resilience mechanism for disaggregated memory. In Chapter 4, we discuss the deployability challenge of memory disaggregation frameworks in public clouds and present Memtrade as a solution to this issue. Chapter 5 discusses TPP – a memory temperature-aware page placement mechanism for heterogeneous tiered-memory systems. We finally conclude the dissertation and discuss future work in Chapter 6.

CHAPTER 2

Effectively Prefetching Remote Memory with Leap

2.1 Introduction

Modern data-intensive applications [84, 56, 161, 162] experience significant performance loss when their complete working sets do not fit into the main memory. At the same time, despite significant and disproportionate memory underutilization in large clusters [319, 264], memory cannot be accessed beyond machine boundaries. Such unused, stranded memory can be leveraged by forming a cluster-wide logical memory pool via *memory disaggregation*, improving application-level performance and overall cluster resource utilization [215, 93, 203].

Two broad avenues have emerged in recent years to expose remote memory to memory-intensive applications. The first requires redesigning applications from the ground up using RDMA primitives [178, 142, 250, 265, 317, 218, 107]. Despite its efficiency, rewriting applications can be cumbersome and may not even be possible for many applications [92]. Alternatives rely on well-known abstractions to expose remote memory; e.g., distributed virtual file system (VFS) for remote file access [92] and distributed virtual memory management (VMM) for *remote memory paging* [165, 157, 211, 275, 203].

Because disaggregated remote memory is slower, keeping hot pages in the faster local memory ensures better performance. Colder pages are moved to the far/remote memory as needed [88, 203, 165]. Subsequent accesses to those cold pages go through a slow data path inside the kernel – for instance, our measurements show that an average 4KB remote page access takes close to 40 μs in state-of-the-art memory disaggregation systems like Infiniswap. Such high access latency significantly affects performance because memory-intensive applications can tolerate at most single μs latency [157, 203]. Note that the latency of existing systems is many times more than the 4.3 μs average latency of a 4KB RDMA operation, which itself can be too high for some applications.

In this paper, we take the following position: *an ideal solution should minimize remote memory accesses in its critical path as much as possible*. In this case, a *local page cache* can reduce the total number of remote memory accesses – a cache *hit* results in a sub-μs latency, comparable to

that of a local page access. An effective prefetcher can proactively bring in correct pages into the cache and increase the cache hit rate.

Unfortunately, existing prefetching algorithms fall short for several reasons. First, they are designed to reduce disk access latency by prefetching sequential disk pages in large batches. Second, they cannot distinguish accesses from different applications. Finally, they cannot quickly adapt to temporal changes in page access patterns within the same process. As a result, being optimistic, they pollute the cache with unnecessary pages. At the same time, due to their rigid pattern detection technique, they often fail to prefetch the required pages into the cache before they are accessed.

In this study, we propose Leap, an online prefetching solution that minimizes the total number of remote memory accesses in the critical path. Unlike existing prefetching algorithms that rely on strict pattern detection, Leap relies on approximation. Specifically, it builds on the Boyer-Moore majority vote algorithm [219] to efficiently identify remote memory access patterns for each individual process. Relying on an approximate mechanism instead of looking for trends in strictly consecutive accesses makes Leap resilient to short-term irregularities in access patterns (e.g., due to multi-threading). It also allows Leap to perform well by detecting trends only from remote page accesses instead of tracing the full virtual memory footprint of an application, which demands continuous scanning and logging of the hardware access bits of the whole virtual address space and results in high CPU and memory overhead. In addition to identifying the majority access pattern, Leap determines how many pages to prefetch following that pattern to minimize cache pollution.

While reducing cache pollution and increasing the cache hit rate, Leap also ensures that the host machine faces minimal memory pressure due to the prefetched pages. To move pages from local to remote memory, the kernel needs to scan through the entire memory address-space to find eviction candidates – the more pages it has, the more time it takes to scan. This increases the memory allocation time for new pages. Therefore, alongside a background LRU-based asynchronous page eviction policy, Leap eagerly frees up a prefetched cache just after it gets hit and reduces page allocation wait time.

We complement our algorithm with an efficient data path design for remote memory accesses that is used in case of a cache *miss*. It isolates per-application remote traffic and cuts inessentials in the end-host software stack (e.g., the block layer) to reduce host-side latency and handle a cache miss with latency close to that of the underlying RDMA operations.

Contribution Overall, we make the following contributions:

- We analyze the data path latency overheads for disaggregated memory systems and find that existing data path components can not consistently support single μs 4KB page access latency (§4.2).

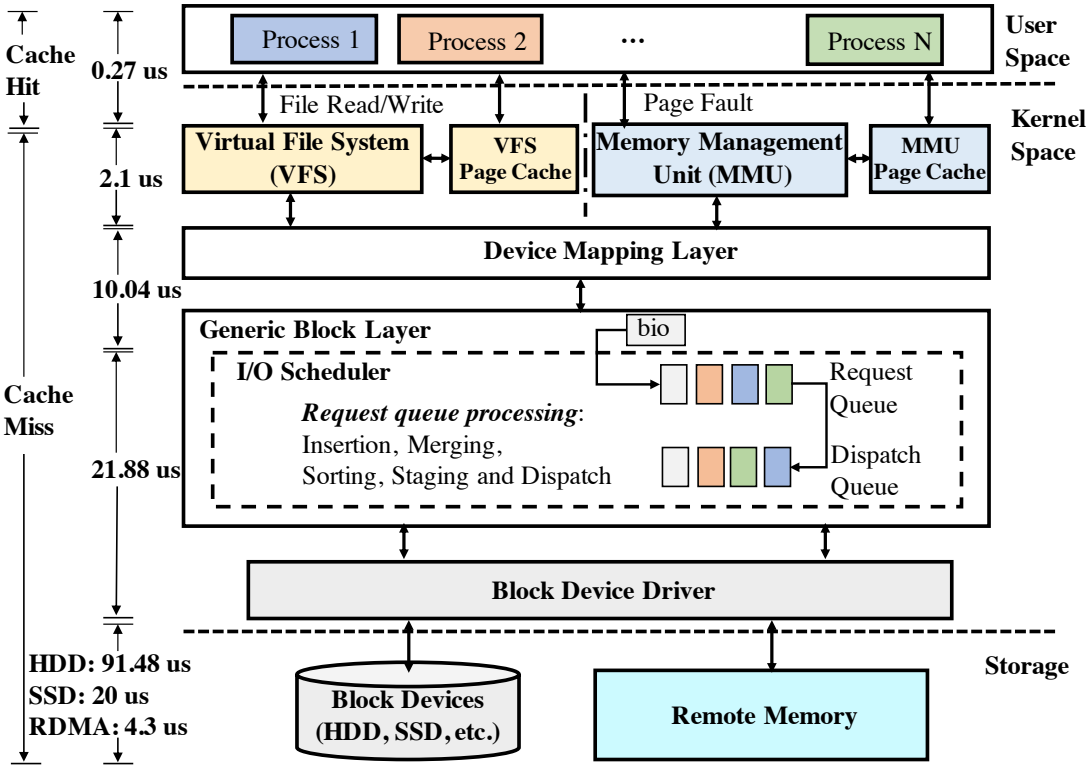


Figure 2.1: High-level life cycle of page requests in Linux data path along with the average time spent in each stage.

- We propose Leap, a novel online prefetching algorithm (§2.3) and an eager prefetch cache eviction policy along with a leaner data path, to improve remote I/O latency.
- We implement Leap on Linux Kernel 4.4.125 as a separate data path for remote memory access (§3.4). Applications can choose either Linux's default data path for traditional usage or Leap for going beyond the machine's boundary using unmodified Linux ABIs.
- We evaluate Leap's effectiveness for different memory disaggregation frameworks. Leap's faster data path and effective cache management improve the median and tail 4KB page access latency by up to 104.04× and 22.62× for micro-benchmarks (§2.5.1) and by 1.27–10.16× for real-world memory-intensive applications with production workloads (§2.5.3).
- We evaluate Leap's prefetcher against practical real-time prefetching techniques (Next-K Line, Stride, Linux Read-ahead) and show that simply replacing the default Linux prefetcher with Leap's prefetcher can provide application-level performance benefit (1.1–3.36× better) even when they are paging to slower storage (e.g., HDD, SSD) (§2.5.2).

2.2 Background and Motivation

2.2.1 Remote Memory

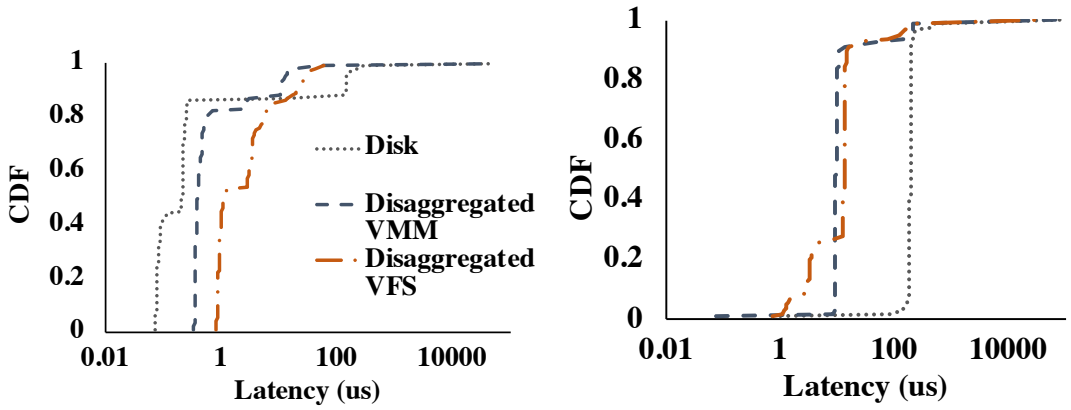
Memory disaggregation systems logically expose unused cluster memory as a global memory pool that is used as the slower memory for machines with extreme memory demand. This improves the performance of memory-intensive applications that have to frequently access slower memory in memory-constrained settings. At the same time, the overall cluster memory usage gets balanced across the machines, decreasing the need for memory over-provisioning per machine.

Access to remote memory over RDMA without significant application rewrites typically relies on two primary mechanisms: disaggregated VFS [92], that exposes remote memory as files and disaggregated VMM for remote memory paging [165, 275, 203]. In both cases, data is communicated in small chunks or pages. In case of remote memory as files, pages go through the file system before they are written to/read from the remote memory. For remote memory paging and distributed OS, page faults cause the virtual memory manager to write pages to and read them from the remote memory.

2.2.2 Remote Memory Data Path

State-of-the-art memory disaggregation frameworks depend on the existing kernel data path that is optimized for slow disks. Figure 2.1 depicts the major stages in the life cycle of a page request. Due to slow disk access times – average latencies for HDDs and SSDs range between 4–5 ms and 80–160 μs, respectively – frequent disk accesses have a severe impact on application throughput and latency. Although the recent rise of memory disaggregation is fueled by the hope that RDMA can consistently provide single μs 4KB page access latency [93, 165, 157], this is often a wishful thinking in practice [322]. Blocking on a page access – be it from HDD, SSD, or remote memory – is often unacceptable.

To avoid blocking on I/O, race conditions, and synchronization issues (e.g., accessing a page while the page out process is still in progress), the kernel uses a page cache. To access a page from slower memory, it is first looked up in the appropriate cache location; a *hit* results in almost memory-speed page access latency. However, when the page is not found in the cache (i.e., a *miss*), it is accessed through a costly block device I/O operation that includes several queuing and batching stages to optimize disk throughput by merging multiple contiguous smaller disk I/O requests into a single large request. On average, these batching and queuing operations cost around 34 μs and over a few milliseconds at the tail. As a result, a cache miss leads to more than $100\times$ slower latency than a hit; it also introduces high latency variations. For microsecond-latency RDMA environments, this unnecessary wait-time has a severe impact on application performance.



(a) **Sequential** (b) **Stride-10**

Figure 2.2: Data path latencies for two access patterns. Memory disaggregation systems have some constant implementation overheads that cap their minimum latency to around 1 μs.

2.2.3 Prefetching in Linux

Linux tries to store files on the disk in adjacent sectors to increase sequential disk accesses. The same happens for paging. Naturally, existing prefetching mechanisms are designed assuming a sequential data layout. The default Linux prefetcher relies on the last two page faults: if they are for consecutive pages, it brings in several sequential pages into the page cache; otherwise, it assumes that there are no patterns and reduces or stops prefetching. This has several drawbacks. First, whenever it observes two consecutive paging requests for consecutive pages, it over-optimistically brings in pages that may not even be useful. As a result, it wastes I/O bandwidth and causes *cache pollution* by occupying valuable cache space. Second, simply assuming the absence of any pattern based on the last two requests is over-pessimistic. Furthermore, all the applications share the same swap space in Linux; hence, pages from two different processes can share consecutive places in the swap area. An application can also have multiple, inter-leaved stride patterns – for example, due to multiple concurrent threads. Overall, considering only the last two requests to prefetch a batch of pages falter on both respects.

To illustrate this, we measure the page access latency for two memory access patterns: (a) **Sequential** accesses memory pages sequentially, and (b) **Stride-10** accesses memory in strides of 10 pages. In both cases, we use a simple application with its working set size set to 2GB. For disaggregated VMM, it is provided 1GB memory to ensure that 50% of its access cause paging. For disaggregated VFS, it performs 1GB remote write and then another 1GB remote read operations.

Figure 2.2 shows the latency distributions for 4KB page accesses from disk and disaggregated remote memory for both of the access patterns. For a prefetch size of 8 pages, both perform well for the **Sequential** pattern; this is because 80% of the requests hit the cache. In contrast,

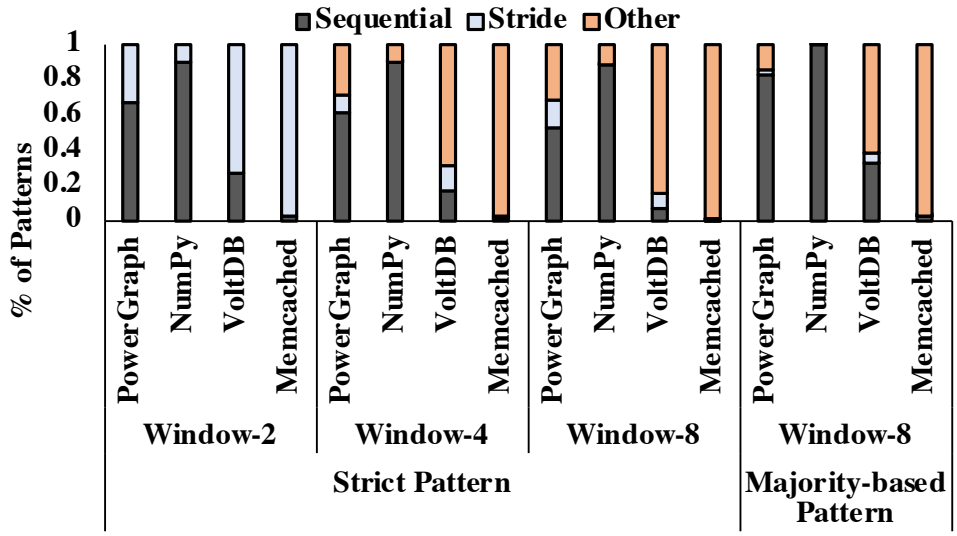


Figure 2.3: Fractions of sequential, stride, and other access patterns in page fault sequences of length X (Window-X).

we observe significantly higher latency in the **Stride-10** case because all the requests miss the page cache due to the lack of consecutiveness in successive page accesses. By analyzing the latency breakdown inside the data path for **Stride-10** (as shown in Figure 2.1), we make two key observations. First, although RDMA can provide significantly lower latency than disk (4.3μs vs. 91.5μs), RDMA-based solutions do not benefit as much from that (38.3μs vs. 125.5μs). This is because of the significant data path overhead (on average 34μs) to prepare and batch a request before dispatching it. Significant variations in the preparation and batching stages of the data path cause the average to stray far from the median. Second, the existing sequential data layout-based prefetching mechanism fails to serve the purpose in the presence of diverse remote page access patterns. Solutions based on fixed stride sizes also fall short because stride sizes can vary over time within the same application. Besides, there can be more complicated patterns beyond stride or no repetitions at all.

Shortcoming of Strict Pattern Finding for Prefetching Figure 2.3 presents the remote page access patterns of four memory-intensive applications during page faults when they are run with 50% of their working sets in memory (more details in Section 2.5.3). Here, we consider all page fault sequences within a window of size $X \in \{2, 4, 8\}$ in these applications. Therefore, we divide the page fault scenarios into three categories: *sequential* when all pages within the window of X are sequential pages, *stride* when the pages within the window of X have the same stride from the first page, and *other* when it is neither sequential nor stride.

The default prefetcher in Linux finds strict sequential patterns in window size $X = 2$ and tunes

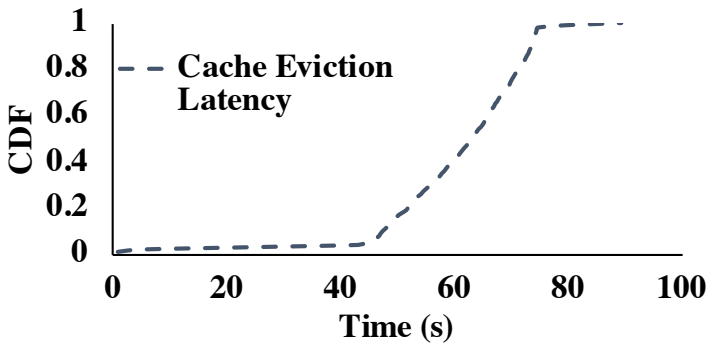


Figure 2.4: Due to Linux's lazy cache eviction policy, page caches waste the cache area for significant amount of time.

up its aggressiveness accordingly. For example, page faults in PowerGraph and VoltDB follow 67% and 27% sequential pattern within window size $X = 2$, respectively. Consequently, for these two applications, Linux optimistically prefetches many pages into the cache. However, if we look at the $X = 8$ case, the percentage of sequential pages within consecutive page faults goes down to 53% and 7% for PowerGraph and VoltDB, respectively. Meaning, for these two applications, 14–20% of the prefetched pages are not consumed immediately. This creates unnecessary memory pressure and might even lead to cache pollution. At the same time, all non-sequential patterns in the $X = 2$ case fall under the stride category. Considering the low cache hit rate, Linux pessimistically decreases/stops prefetching in those cases, which leads to a stale page cache.

Note that strictly expecting all X accesses to follow the same pattern results in not having any patterns at all (e.g., when $X = 8$), because this cannot capture the transient interruptions in sequence. In that case, following the major sequential and/or stride trend within a limited page access history window is more resilient to short-term irregularities. Consecutively, when $X = 8$, a majority-based pattern detection can detect 11.3%–29.7% more sequential accesses. Therefore, it can successfully prefetch more accurate pages into the page cache. Besides sequential and stride access patterns, it is also transparent to irregular access patterns; e.g., for Memcached, it can detect 96.4% of the irregularity.

Prefetch Cache Eviction Linux kernel maintains an asynchronous background thread (`kswapd`) to monitor the machine's memory consumption. If the overall memory consumption goes beyond a critical memory pressure or a process's memory usage hits its limit, it determines the eviction candidates by scanning over the in-memory pages to find out the least-recently-used (LRU) ones. Then, it frees up the selected pages from the main memory to allocate new pages. A prefetched cache waits into the LRU list for its turn to get selected for eviction even though it has already been used by a process (Figure 2.4). Unnecessary pages waiting for eviction in-memory

	Low Computational Complexity	Low Memory Overhead	Unmodified Application	HW/SW Independent	Temporal Locality	Spatial Locality	High Prefetch Utilization
Next-N-Line [232]	✓	✓	✓	✓	X	✓	X
Stride [106]	✓	✓	✓	✓	X	✓	X
GHB PC [240]	X	X	✓	X	✓	✓	✓
Instruction Prefetch [152, 194]	X	X	X	X	✓	✓	✓
Linux Read-Ahead [302]	✓	✓	✓	✓	✓	✓	X
Leap Prefetcher	✓	✓	✓	✓	✓	✓	✓

Table 2.1: Comparison of prefetching techniques based on different objectives.

leads to extra scanning time. This extra wait-time due to lazy cache eviction policy adds to the overall latency, especially in a high memory pressure scenario.

2.3 Remote Memory Prefetching

In this section, we first highlight the characteristics of an ideal prefetcher. Next, we present our proposed online prefetcher along with its different components and the design principles behind them. Finally, we discuss the complexity and correctness of our algorithm.

2.3.1 Properties of an Ideal Prefetcher

A prefetcher's effectiveness is measured along three axes:

- *Accuracy* is the ratio of total cache hits and total pages added to the cache via prefetching.
- *Coverage* measures the ratio of the total cache hit from the prefetched pages and the total number of requests (e.g., page faults in case of remote memory paging solutions).
- *Timeliness* of an accurately prefetched page is the time gap from when it was prefetched to when it was first hit.

Trade-off An aggressive prefetcher can hide the slower memory access latency by bringing pages well ahead of the access requests. This might increase the accuracy, but as prefetched pages wait longer to get consumed, this wastes the effective cache and I/O bandwidth. On the other hand, a conservative prefetcher has lower prefetch consumption time and reduces cache and bandwidth contention. However, it has lower coverage and cannot hide memory access latency completely. An effective prefetcher must balance all three.

An effective prefetcher must be adaptive to temporal changes in memory access patterns as well. When there is a predictable access pattern, it should bring pages aggressively. In contrast, during irregular accesses, the prefetch rate should be throttled down to avoid cache pollution.

Prefetching algorithms use prior page access information to predict future access patterns. As such, their effectiveness largely depends on how well they can detect patterns and predict. A real-time prefetcher has to face a trade-off between pattern identification accuracy vs. computational complexity and resource overhead. High CPU usage and memory consumption will negatively impact application performance even though they may help in increasing accuracy.

Common Prefetching Techniques The most common and simple form of prefetching is spatial pattern detection [227]. Some specific access patterns (i.e., stride, stream, etc.) can be detected with the help of special hardware (HW) features [173, 177, 279, 330]. However, they are typically applied to identify patterns in instruction access that are more regular; in contrast, data access patterns are more irregular. Special prefetch instructions can also be injected into an application's source code, based on compiler or post-execution based analysis [253, 188, 256, 152, 194]. However, compiler-injected prefetching needs a static analysis of the cache miss behavior before the application runs. Hence, they are not adaptive to dynamic cache behavior. Finally, HW- or software (SW)-dependent prefetching techniques are limited to the availability of the special HW/SW features and/or application modification.

Summary An ideal prefetcher should have low computational and memory overhead. It should have high accuracy, coverage, and timeliness to reduce cache pollution; an adaptive prefetch window is imperative to fulfill this requirement. It should also be flexible to both spatial and temporal locality in memory accesses. Finally, HW/SW independence and application transparency make it more generic and robust.

Table 2.1 compares different prefetching methods.

2.3.2 Majority Trend-Based Prefetching

Leap has two main components: *detecting trends* and *determining what to prefetch*. The first component looks for any approximate trend in earlier accesses. Based on the trend availability and prefetch utilization information, the latter decides how many and which pages to prefetch.

2.3.2.1 Trend Detection

Existing prefetch solutions rely on strict pattern identification mechanisms (e.g., sequential or stride of fixed size) and fail to ignore temporary irregularities. Instead, we consider a relaxed approach that is robust to short-term irregularities. Specifically, we identify the majority Δ values in a fixed-size (H_{size}) window of remote page accesses (ACCESSHISTORY) and ignore the rest. For a window of size w, a Δ value is said to be the major only if it appears at least $\lfloor w/2 \rfloor + 1$

Algorithm 1 Trend Detection

```
1:  procedure FINDTREND(N_split)
2:      H_size ← SIZE(AccessHistory)
3:      w ← H_size/N_split                          ▷ Start with small detection window
4:      Δ_maj ← ∅
5:      while true do
6:          Δ_maj ← Boyer-Moore on {H_head, ..., H_head−w−1}
7:          w ← w * 2
8:          if Δ_maj ≠ major trend then
9:              Δ_maj ← ∅
10:         if Δ_maj ≠ ∅ or w > H_size then
11:             return Δ_maj
12:     return Δ_maj
```

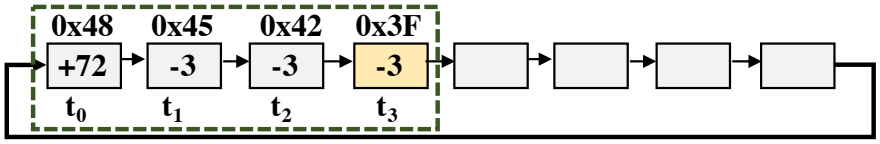
times within that window. To find the majority Δ, we use the Boyer-Moore majority vote algorithm [219] (Algorithm 1), a linear-time and constant-memory algorithm, over ACCESSHISTORY elements. Given a majority Δ, due to the temporal nature of remote page access events, it can be hypothesized that subsequent Δ values are more likely to be the same as the majority Δ.

Note that if two pages are accessed together, they will be aged and evicted together in the slower memory space at contiguous or nearby addresses. Consequently, the temporal locality in virtual memory accesses will also be observed in the slower page accesses and an approximate stride should be enough to detect that.

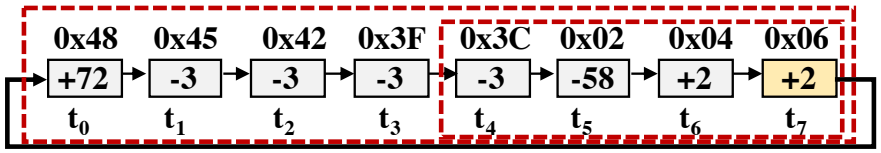
Window Management If a memory access sequence follows a regular trend, then the majority Δ is likely to be found in almost any part of that sequence. In that case, a smaller window can be more effective as it reduces the total number of operations. So instead of considering the entire ACCESSHISTORY, we start with a smaller window that starts from the head position (H_{head}) of ACCESSHISTORY. For a window of size w, we find the major Δ appearing in the $H_{head}, H_{head-1}, ..., H_{head-w-1}$ elements.

However, in the presence of short-term irregularities, small windows may not detect a majority. To address this, the prefetcher starts with a small detection window and doubles the window size up to ACCESSHISTORY size until it finds a majority; otherwise, it determines the absence of a majority. The smallest window size can be controlled by N_{split}.

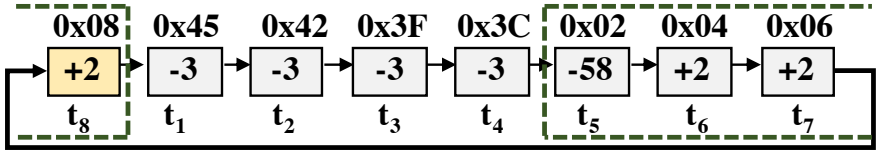
Example Let us consider a ACCESSHISTORY with $H_{size} = 8$ and $N_{split} = 2$. Say pages with the following addresses: `0x48`, `0x45`, `0x42`, `0x3F`, `0x3C`, `0x02`, `0x04`, `0x06`, `0x08`, `0x0A`, `0x0C`, `0x10`, `0x39`, `0x12`, `0x14`, `0x16`, were requested in that order. Figure 2.5 shows the corresponding Δ values stored in ACCESSHISTORY, with t_0 being the earliest and t_{15} being the



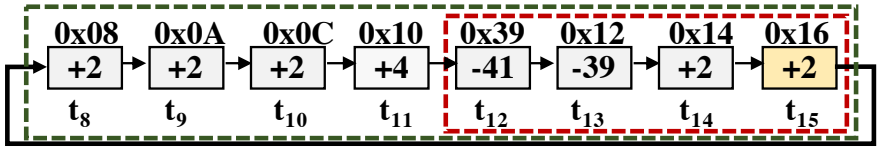
(a) at time t_3



(b) at time t_7



(c) at time t_8



(d) at time t_{15}

Figure 2.5: Content of ACCESSHISTORY at different time. Solid colored boxes indicate the head position at time t_i. Dashed boxes indicate detection windows. Here, time rolls over at t_8.

latest request. At t_i, H_{head} stays at the t_i-th slot.

FINDTREND in Algorithm 1 will initially try to detect a trend using a window size of 4. Upon failure, it will look for a trend first within a window size of 8.

At time t_3, FINDTREND successfully finds a trend of -3 within the t_0–t_3 window (Figure 2.5a).

At time t_7, the trend starts to shift from -3 to +2. At that time, t_4–t_7 window does not have a majority Δ, which doubles the window to consider t_0–t_7. This window does not have any majority Δ either (Figure 2.5b). However, at t_8, a majority Δ of +2 within t_5–t_8 will be adopted as the new trend (Figure 2.5c).

Similarly, at t_{15}, we have a majority of +2 in the t_8–t_{15}, which will continue to the +2 trend found at t_8 while ignoring the short-term variations at t_{12} and t_{13} (Figure 2.5d).

2.3.2.2 Prefetch Candidate Generation

So far we have focused on identifying the presence of a trend. Algorithm 2 determines whether and how to use that trend for prefetching for a request for page P_t.

Algorithm 2 Prefetch Candidate Generation

procedure GETPREFETCHWINDOWSIZE(page P_t)
    PW_{size_t} ▷ Current prefetch window size
    $PW_{size_{t-1}}$ ▷ Last prefetch window size
    C_{hit} ▷ Prefetched cache hits after last prefetch
    if $C_{hit} = 0$ **then**
        if P_t follows the current trend **then**
            $PW_{size_t} \leftarrow 1$ ▷ Prefetch a page along trend
        else
            $PW_{size_t} \leftarrow 0$ ▷ Suspend prefetching
    else ▷ Earlier prefetches had hits
        $PW_{size_t} \leftarrow$ Round up $C_{hit} + 1$ to closest power of 2
    $PW_{size_t} \leftarrow \mathbf{min}(PW_{size_t}, PW_{size_{max}})$
    if $PW_{size_t} < PW_{size_{t-1}}/2$ **then** ▷ Low cache hit
        $PW_{size_t} \leftarrow PW_{size_{t-1}}/2$ ▷ Shrink window smoothly
    $C_{hits} \leftarrow 0$
    $PW_{size_{t-1}} \leftarrow PW_{size_t}$
    return PW_{size_t}

procedure DOPREFETCH(page P_t)
    $PW_{size_t} \leftarrow$ GETPREFETCHWINDOWSIZE(P_t)
    if $PW_{size_t} \neq 0$ **then**
        $\Delta_{maj} \leftarrow$ FINDTREND(N_split)
        if $\Delta_{maj} \neq \emptyset$ **then**
            Read PW_{size_t} pages with Δ_{maj} stride from P_t
        else
            Read PW_{size_t} pages around P_t with latest Δ_{maj}
    else
        Read only page P_t

We determine the prefetch window size (PW_{size_t}) based on the accuracy of prefetches between two consecutive prefetch requests (see GETPREFETCHWINDOWSIZE). Any cache hit of the prefetched data between two consecutive prefetch requests indicates the overall effectiveness of the prefetch. In case of high effectiveness (i.e., a high cache hit), PW_{size_t} is expanded until it reaches maximum size ($PW_{size_{max}}$). On the other hand, low cache hit indicates low effectiveness; in that case, the prefetch window size gets reduced. However, in the presence of drastic drops, prefetching is not suspended immediately. The prefetch window is shrunk smoothly to make the algorithm flexible to short-term irregularities. When prefetching is suspended, no extra pages are prefetched until a new trend is detected. This is to avoid cache pollution during irregular/unpredictable accesses.

Given a non-zero PW_{size}, the prefetcher brings in PW_{size} pages following the current trend, if any (DOPREFETCH). If no majority trend exists, instead of giving up right away, it speculatively brings PW_{size} pages around P_t's offset following the previous trend. This is to ensure that short-term irregularities cannot completely suspend prefetching.

Prefetching in the Presence of Irregularity FINDTREND can detect a trend within a window of size w in the presence of at most $\lfloor w/2 \rfloor - 1$ irregularities within it. If the window size is too small or the window has multiple perfectly interleaved threads with different strides, FINDTREND will consider it a random pattern. In that case, if the PW_{size} has a non-zero value then it performs a speculative prefetch (line 25) with the previous Δ_{maj}. If that Δ_{maj} is one of the interleaved strides, then this speculation will cause cache hit and continue. Otherwise, PW_{size} will eventually be zero and the prefetcher will stop bringing unnecessary pages. In that case, the prefetcher cannot be worse than the existing prefetch algorithms.

Prefetching During Constrained Bandwidth In Leap, faulted page read and prefetch are done asynchronously. Here, prefetching has a lower priority. In extreme bandwidth constraints, prefetched pages arrive late and result in fewer cache hits. This will eventually shrink down PW_{size}. Thus, dynamic prefetch window sizing helps in bandwidth-constrained scenarios.

Effect of Huge Page Linux kernel splits a huge page into 4KB pages before swapping. When transparent huge page is enabled, Leap will be applied on these splitted 4KB pages.

Using huge pages will result in high amplification for dirty data [112]. Besides, average RDMA latencies for 4KB vs 2MB page are 3μs vs 330μs. If huge pages were never split, to maintain single μs latency for 2MB pages, we need a significantly larger prefetch window size ($PW_{size} \geq 128$), demanding more bandwidth and cache space, and making mispredictions more expensive.

2.3.3 Analysis

Time Complexity The FINDTREND function in Algorithm 1 initially tries to detect trend aggressively within a smaller window using the Boyer-Moor Majority Voting algorithm. If it fails, then it expands the window size. The Boyer-Moor Majority Voting algorithm (line 6) detects a majority element (if any) in $O(w)$ time, where w is the size of the window. In the worst case, it will invoke the Boyer-Moor Majority Voting algorithm for $O(logH_{size})$ times. However, as the windows are continuous, searching in a new window does not need to start from the beginning and the algorithm never access the same item twice. Hence, the worst-case time complexity of the FINDTREND function is $O(H_{size})$, where H_{size} is the size of the ACCESSHISTORY queue.

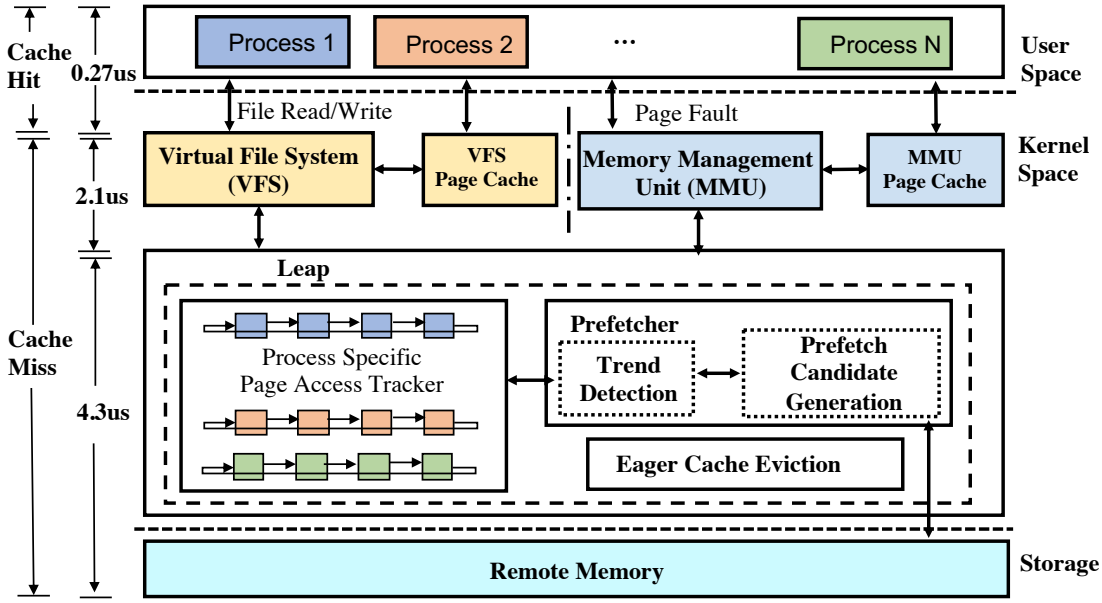


Figure 2.6: Leap has a faster data path for a cache miss.

For smaller H_{size} the computational complexity is constant. Even for $H_{size} = 32$, the prefetcher provides significant performance gain (§2.5) that outweighs the slight extra computational cost.

Memory Complexity The Boyer-Moor Majority Voting algorithm operates on constant memory space. FINDTREND just invokes the Boyer-Moor Majority Voting algorithm and does not require any additional memory to execute. So, the Trend Detection algorithm needs O(1) space to operate.

Correctness of Trend Detection The correctness of FINDTREND depends on that of the Boyer-Moor Majority Voting algorithm, which always provides the majority element, if one exists, in linear time (see [219] for the formal proof).

2.4 System Design

We have implemented our prefetching algorithm as a data path replacement for memory disaggregation frameworks (we refer to this design as Leap data path) alongside the traditional data path in Linux kernel v4.4.125. Leap has three primary components: a page access tracker to isolate processes, a majority-based prefetching algorithm, and an eager cache eviction mechanism. All of them work together in the kernel space to provide a faster data path. Figure 4.4 shows the basic architecture of Leap's remote memory access mechanism. It takes only around 400 lines of code to implement the page access tracker, prefetcher, and the eager eviction mechanism.

2.4.1 Page Access Tracker

Leap isolates each process's page access data paths. The page access tracker monitors page accesses inside the kernel that enables the prefetcher to detect application-specific page access trends. Leap does not monitor in-memory pages (hot pages) because continuously scanning and recording the hardware access bits of a large number of pages causes significant computational overhead and memory consumption. Instead, it monitors only the cache look-ups and records the access sequence of the pages after I/O requests or page faults, trading off a small loss in access pattern detection accuracy for low resource overhead. As temporal locality in the virtual memory space results in a spatial locality in the remote address space, just monitoring the remote page accesses is often enough.

The page access tracker is added as a separate control unit inside the kernel. Upon a page fault, during the page-in operation (`do_swap_page()` under `mm/memory.c`), we notify (`log_access_history()`) Leap's page access tracker about the page fault and the process involved. Leap maintains process-specific fixed-size (H_{size}) FIFO ACCESSHISTORY circular queues to record the page access history. Instead of recording exact page addresses, however, we only store the difference between two consecutive requests (Δ). For example, if page faults happen for addresses `0x2`, `0x5`, `0x4`, `0x6`, `0x1`, `0x9`, then ACCESSHISTORY will store the corresponding Δ values: 0, +3, -1, +2, -5, +8. This reduces the storage space and computation overhead during trend detection (§2.3.2.1).

2.4.2 The Prefetcher

To increase the probability of cache hit, Leap incorporates the majority trend-based prefetching algorithm (§2.3.2). Here, the prefetcher considers each process's earlier remote page access histories available in the respective ACCESSHISTORY to efficiently identify the access behavior of different processes. Because threads of the same process share memory with each other, we choose process-level detection over thread-based. Thread-based pattern detection may result in requesting the same page for prefetch multiple times for different threads.

Two consecutive page access requests are temporally correlated in the sense that they may happen together in the future. The Δ values stored in the ACCESSHISTORY records the spatial locality in the temporally correlated page accesses. Therefore, the prefetcher utilizes both temporal and spatial localities of page accesses to predict future page demand.

The prefetcher is added as a separate control unit inside the kernel. While paging-in, instead of going through the default `swapin_readahead()`, we re-route it through the prefetcher's `do_prefetch()` function. Whenever the prefetcher generates the prefetch candidates, Leap bypasses the expensive request scheduling and batch-

ing operations of the block layer (`swap_readpage()`/`swap_writepage()` for paging and `generic_file_read()`/`generic_file_write()` for the file systems) and invokes `leap_remote_io_request()` to re-direct the request through Leap's asynchronous remote I/O interface over RDMA (§2.4.4).

2.4.3 Eager Cache Eviction

Leap maintains a circular linked list of prefetched caches (PREFETCHFIFOLRULIST). Whenever a page is fetched from remote memory, besides the kernel's global LRU lists, Leap adds it at the tail of the linked list. After the prefetch cache gets hit and the page table is updated, Leap marks the page as an eviction candidate. A separate background process continuously removes eviction candidates from PREFETCHFIFOLRULIST and frees up those pages to the buddy list. As an accurate prefetcher is timely in using the prefetched data, in Leap, prefetched caches do not wait long to be freed up. For workloads where repeated access to paged-in data is not so common, this eager eviction of prefetched pages reduces the wait time to find and allocate new pages - on average, page allocation time is reduced by $750ns$ (36% less than the usual). Thus, new pages can be brought to the memory more quickly leading to a reduction in the overall data path latency. For workloads where paged-in data is repeatedly used, Leap considers the frequency of access for prefetched pages and exempt them from eager eviction.

However, if the prefetched pages need to be evicted even before they get consumed (e.g., at severe global memory pressure or extreme constrained prefetch cache size scenario), due to the lack of any access history, prefetched pages will follow a FIFO eviction order among themselves from the PREFETCHFIFOLRULIST. Reclamation of other memory (file-backed or anonymous page) follows the existing LRU-based eviction technique by `kswapd` in the kernel. We modify the kernel's Memory Management Unit (`mm/swap_state.c`) to add the prefetch eviction related functions.

2.4.4 Remote I/O Interface

Similar to existing works [165, 92], Leap uses an agent in each host machine to expose a remote I/O interface to the VFS/VMM over RDMA. The host machine's agent communicates to another remote agent with its resource demand and performs remote memory mapping. The whole remote memory space is logically divided into fixed-size memory slabs. A host agent can map slabs across one or more remote machine(s) according to its resource demand, load balancing, and fault tolerance policies.

The host agent maintains a per CPU core RDMA connection to the remote agent. We use the multi-queue IO queuing mechanism where each CPU core is configured with an individual RDMA dispatch queue for staging remote read/write requests. Upon receiving a remote I/O request, the

host generates/retrieves a slot identifier, extracts the remote memory address for the page within that slab, and forwards the request to the RDMA dispatch queue to perform read/write over the RDMA NIC. During the whole process, Leap completely bypasses block layer operations.

Resilience, Scalability, & Load Balancing One can use existing memory disaggregation frameworks [165, 275, 92] with respective scalability and fault tolerance characteristics and still have the performance benefits of Leap. We do not claim any innovation here. In our implementation, the host agent leverages the power of two choices [233] to minimize memory imbalance across remote machines. Remote in-memory replication is the default fault tolerance mechanism in Leap.

2.5 Evaluation

We evaluate Leap on a 56 Gbps InfiniBand cluster on CloudLab [25]. Our key results are as follows:

- Leap provides a faster data path to remote memory. Latency for 4KB remote page accesses improves by up to $104.04\times$ ($24.96\times$) at the median and $22.06\times$ ($17.32\times$) at the tail in case of Disaggregated VMM (VFS) (§2.5.1).
- While paging to disk, our prefetcher outperforms its counterparts (Next-K, Stride, and Read-Ahead) by up to $1.62\times$ for cache pollution and up to $10.47\times$ for cache miss. It improves prefetch coverage by up to 37.51% (§2.5.2).
- Leap improves the end-to-end application completion times of PowerGraph, NumPy, VoltDB, and Memcached by up to $9.84\times$ and their throughput by up to $10.16\times$ over existing memory disaggregation solutions (§2.5.3).

Methodology We integrate Leap inside the Linux kernel, both in its VMM and VFS data paths. As a result, we evaluate its impact on three primary mediums.

- *Local disks*: Here, Linux swaps to a local HDD and SSD.
- *Disaggregated VMM (D-VMM)*: To evaluate Leap's benefit for disaggregated VMM system, we integrate Leap with the latest commit of Infiniswap on GitHub [47].
- *Disaggregated VFS (D-VFS)*: To evaluate Leap's benefit for a disaggregated VFS system, we add Leap to our implementation of Remote Regions [92], which is not open-source.

For both of the memory disaggregation systems, we use respective load balancing and fault tolerance mechanisms. Unless otherwise specified, we use ACCESSHISTORY buffer size H_{size} = 32, and maximum prefetch window size $PW_{size_{max}}$ = 8.

Each machine in our evaluation has 64 GB of DRAM and $2\times$ Intel Xeon E5-2650v2 with 16 cores (32 hyperthreads).

2.5.1 Microbenchmark

We start by analyzing Leap's latency characteristics with the two simple access patterns described in Section 4.2.

During sequential access, due to prefetching, 80% of the total page requests hit the cache in the default mechanism. On the other hand, during stride access, all prefetched pages brought in by the Linux prefetcher are unused and every page access request experiences a cache miss.

Due to Leap's faster data path, for **Sequential**, it improves the median by $4.07\times$ and 99^{th} percentile by $5.48\times$ for disaggregated VMM (Figure 2.7a). For **Stride-10**, as the prefetcher can detect strides efficiently, Leap performs almost as good as it does during the sequential accesses. As a result, in terms of 4KB page access latency, Leap improves disaggregated VMM by $104.04\times$ at the median and $22.06\times$ at the tail (Figure 2.7b).

Leap provides similar performance benefits during memory disaggregation through the file abstraction as well. During sequential access, Leap improves 4KB page access latency by $1.99\times$ at the median and $3.42\times$ at the 99^{th} percentile. During stride access, the median and 99^{th} percentile latency improves by $24.96\times$ and $17.32\times$, respectively.

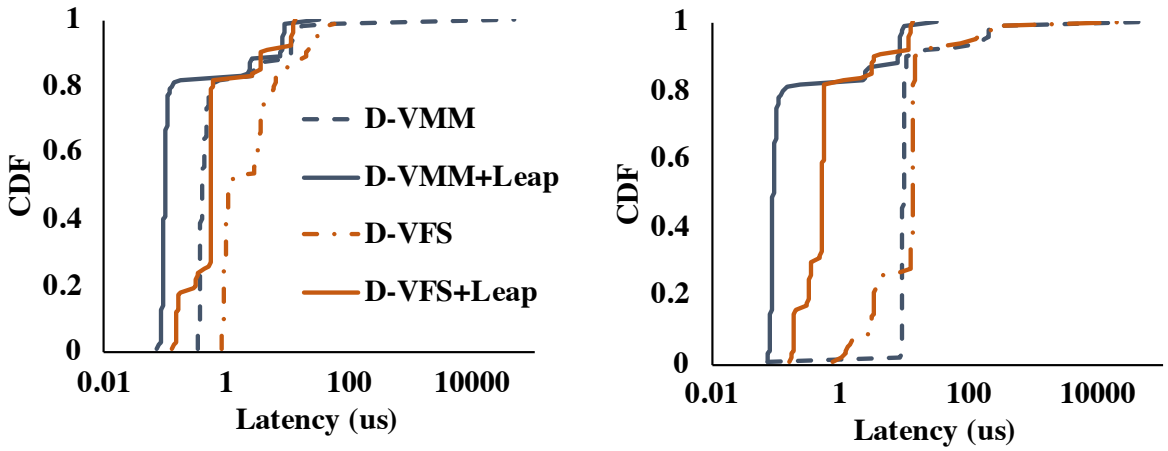
Performance Benefit Breakdown For disaggregated VMM (VFS), the prefetcher improves the 99^{th} percentile latency by 25.4% (23.1%) over the optimized data path where Leap's eager cache eviction contributes another 9.7% (8.5%) improvement.

As the idea of using far/remote memory for storing cold data is getting more popular these days [88, 203, 165], throughout the rest of the evaluation, we focus only on remote paging through a disaggregated VMM system.

2.5.2 Performance Benefit of the Prefetcher

Here, we focus on the effectiveness of the prefetcher itself. We use four real-world memory-intensive applications and workload combinations (Figure 2.3) used in prior works[165, 92].

- TunkRank[2] on PowerGraph[161] to measure the influence of a Twitter user from the follower graph [199]. This workload has a significant amount of stride, sequential, and random access patterns.



(a) **Sequential** (b) **Stride-10**

Figure 2.7: Leap provides lower 4KB page access latency for both sequential and stride access patterns.

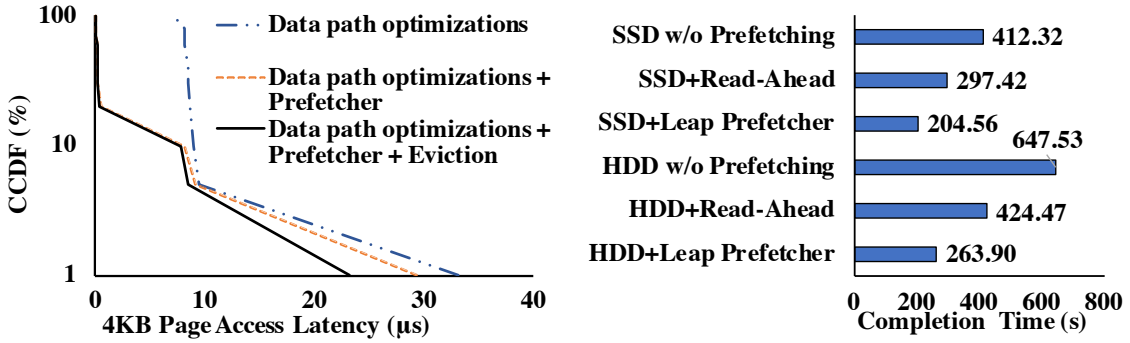
- Matrix multiplication on NumPy[246] over matrices of floating points. This has mostly sequential patterns.
- TPC-C benchmark[79] on VoltDB [84] to simulate an order-entry environment. We set 256 warehouses and 8 sites and run 2 million transactions. This has mostly random with a few amount of sequential patterns.
- Facebook's ETC workload[103] on Memcached[56]. We use 10 million SET operations to populate the Memcached server. Then we perform another 10 million queries (5%SETs, 95%GETs). This has mostly random patterns.

The peak memory usage of these applications varies from 9–38.2 GB. To prompt remote paging, we limit an application's memory usage through `cgroups` [23]. To separate the benefit of the prefetcher, we run all of the applications on disk (with existing block layer-based data path) with 50% memory limit.

2.5.2.1 Prefetch Utilization

We find Leap's prefetcher's benefit over following practical and realtime prefetching techniques:

- *Next-N-Line Prefetcher* [232] aggressively brings N pages sequentially mapped to the page with the cache miss if they are not in the cache.



(a) Benefit Breakdown

(b) Prefetcher with Slow Storage

Figure 2.8: The prefetcher is effective for different storage systems.

	PowerGraph				NumPy				VoltDB				Memcached			
	Next-N-Line	Stride	Read-Ahead	Leap	Next-N-Line	Stride	Read-Ahead	Leap	Next-N-Line	Stride	Read-Ahead	Leap	Next-N-Line	Stride	Read-Ahead	Leap
Cache Add (millions)	4.88	3.88	3.85	**3.01**	10.75	10.52	10.61	**10.08**	6.50	6.23	5.91	**5.20**	4.65	4.14	4.06	**3.25**
Cache Miss (millions)	1.11	1.61	0.26	**0.15**	0.13	0.16	0.14	**0.12**	1.53	2.24	0.96	**0.90**	1.44	1.39	1.36	**0.96**
Completion Time (s)	683.92	885.86	462.54	**263.90**	1410.30	1380.10	1332.40	**1240.60**	2017.47	2454.72	2064.60	**1799.84**	382.54	374.60	366.91	**302.43**
Accuracy (%)	**55.30**	45.60	45.10	44.60	**89.60**	89.40	89.20	88.90	**40.20**	39.50	39.90	37.60	41.80	**42.10**	41.90	39.40
Coverage (%)	70.90	52.30	86.80	**89.80**	95.80	96.30	96.80	**98.60**	61.20	47.40	68.50	**71.00**	51.70	52.40	56.90	**57.60**
Timeliness (ms) - 95^{th} Percentile	19.10	**0.03**	0.39	0.07	10.34	**0.02**	0.24	0.06	22125.14	**34.32**	64314.96	776.68	32417.89	**466.64**	46679.77	886.67

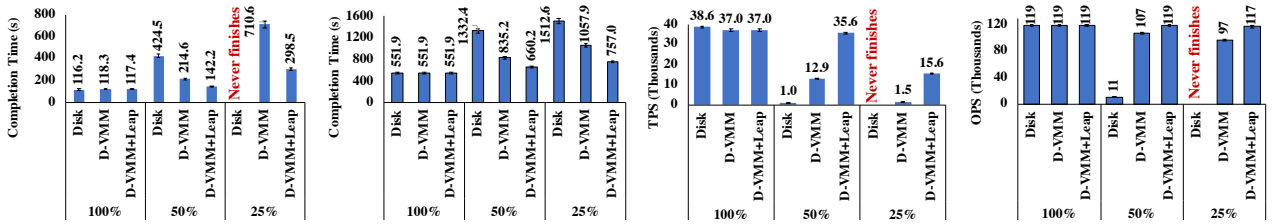
Table 2.2: Leap's prefetcher reduces cache pollution and cache miss events. With higher coverage, better timeliness and almost similar accuracy, the prefetcher outperforms its counterparts in terms of application level performance. Here, shaded numbers indicate the best performances.

- *Stride Prefetcher* [106] brings pages following a stride pattern relative to the current page upon a cache miss. The aggressiveness of this prefetcher depends on the accuracy of the past prefetch.
- *Linux Read-Ahead* prefetches an aligned block of pages containing the faulted page [302]. Linux uses prefetch hit count and an access history of size 2 to control the aggressiveness of the prefetcher.

Impact on the Cache As the volume of data fetched into the cache increases, the prefetch hit rate increases as well. However, thrashing begins as soon as the working set exceeds the cache capacity. As a result, useful demand-fetched pages are evicted. Table 2.2 shows that Leap's prefetcher uses fewer page caches (4.37–62.13%) than the other prefetchers for every workload.

A successful prefetcher reduces the number of cache misses by bringing the most accurate pages into the cache. Leap's prefetcher experiences fewer cache miss events (1.1–10.47×) and enhances the effective usage of the cache space.

Application Performance Due to the improvement in cache pollution and reduction of cache miss, using Leap's prefetcher, all of the applications experience the lowest completion time. Based



(a) PowerGraph Completion Time (b) NumPy Completion Time (c) VoltDB Throughput (d) Memcached Throughput

Figure 2.9: Leap provides lower completion times and higher throughput over Infiniswap's default data path for different memory limits. Note that lower is better for completion time, while higher is better for throughput. Disk refers to HDD in this figure.

on the access pattern, Leap's prefetcher improves the application completion time by 7.4–75.3% over Linux's default Read-Ahead prefetching technique (Table 2.2).

Effectiveness If a prefetcher brings every possible page in the page cache, then it will be 100% accurate. However, in reality, one cannot have an infinite cache space due to large data volumes and/or multiple applications running on the same machine. Besides, optimistically bringing pages may create cache contention, which reduces the overall performance.

Leap's prefetcher trades off cache pollution with comparatively lower accuracy. In comparison to other prefetchers, it shows 0.3–10.8% lower accuracy (Table 2.2). This accuracy loss is linear to the number of cache adds done by the prefetchers. Because the rest of the prefetchers bring in too many pages, their chances of getting lucky hits increase too. Although Leap has the lowest accuracy, its high coverage (0.7–37.5%) allows it to serve with accurate prefetches with a lower cache pollution cost. At the same time, it has an improved timeliness over Read-Ahead (4–52.6×) at the 95^{th} percentile. Due to the higher coverage, better timeliness, and almost similar accuracy, Leap's prefetcher thus outperforms others in terms of application-level performance. Note that despite having the best timeliness, Stride has the worst coverage and completion time that impedes its overall performance.

2.5.2.2 Performance Benefit Breakdown

Figure 2.8a shows the performance benefit breakdown for each of the components of Leap's data path. For PowerGraph at 50% memory limit, due to data path optimizations, Leap provides with single μs latency for 4KB page accesses up to the 95^{th} percentile. Inclusion of the prefetcher ensures sub-μs 4KB page access latency up to the 85^{th} percentile and improves the 99^{th} percentile latency by 11.4% over Leap's optimized data path. The eager eviction policy reduces the page cache allocation time and improves the tail latency by another 22.2%.

2.5.2.3 Performance Benefit for HDD and SSD

To observe the usefulness of the prefetcher for different slow storage systems, we incorporate it into Linux's default data path while paging to SSD. For PowerGraph, Leap's prefetcher improves the overall application run time by $1.45\times$ ($1.61\times$) for SSD (HDD) over Linux's default prefetcher (Figure 2.8b).

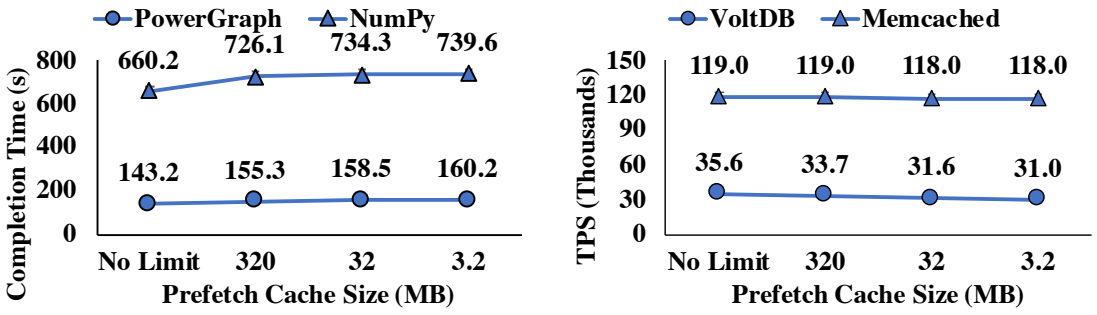
2.5.3 Leap's Overall Impact on Applications

Finally, we evaluate the overall benefit of Leap (including all of its components) for the applications mentioned in Section 2.5.2. We limit an application's memory usage to fit 100%, 50%, 25% of its peak memory usage. Here, we considered the extreme memory constrain (e.g., 25%) to validate the applicability of Leap to recent resource (memory) disaggregation frameworks that operate on a minimal amount of local memory [275].

PowerGraph PowerGraph suffers significantly for cache misses in Infiniswap (Figure 2.9a). In contrast, Leap increases the cache hit rate by detecting 19.03% more remote page access patterns over Read-Ahead. The faster the prefetch cache hit happens, the faster the eager cache eviction mechanism frees up page caches and eventually helps in faster page allocations for a new prefetch. Besides, due to more accurate prefetching, Leap reduces the wastage in both cache space and RDMA bandwidth. This improves 4KB remote page access time by $8.17\times$ and $2.19\times$ at the 99^{th} percentile for 50% and 25% cases, respectively. Overall, the integration of Leap to Infiniswap improves the completion time by $1.56\times$ and $2.38\times$ at 50% and 25% cases, respectively.

NumPy Leap can detect most of the remote page access patterns (10.4% better than Linux's default prefetcher). As a result, similar to PowerGraph, for NumPy, Leap improves the completion time by $1.27\times$ and $1.4\times$ for Infiniswap at 50% and 25% memory limit, respectively (Figure 2.9b). The 4KB page access time improves by $5.28\times$ and $2.88\times$ at the 99^{th} percentile at 50% and 25% cases, respectively.

VoltDB Latency-sensitive applications like VoltDB suffer significantly due to paging. During paging, due to Linux's slower data path, Infiniswap suffers 65.12% and 95.72% lower throughput than local memory behavior at 50% and 25% cases, respectively. In contrast, Leap's better prefetching (11.6% better than Read-Ahead) and instant cache eviction improves the 4KB page access time – $2.51\times$ and $2.7\times$ better 99^{th} percentile at 50% and 25% cases, respectively. However, while executing short random transactions, VoltDB has irregular page access patterns (69% of the total remote page accesses). At that time, our prefetcher's adaptive throttling helps the most by



(a) Completion Time

(b) Throughput

Figure 2.10: Leap has minimal performance drop for Infiniswap even in the presence of O(1) MB cache size.

not congesting the RDMA. Overall, Leap faces smaller throughput loss (3.78% and 57.97% lower than local memory behavior at 50% and 25% cases, respectively). Leap improves Infiniswap's throughput by $2.76\times$ and $10.16\times$ at 50% and 25% cases, respectively (Figure 2.9c).

Memcached This workload has a mostly random remote page access pattern. Leap's prefetcher can detect most of them and avoids prefetching in the presence of randomness. This results in fewer remote requests and less cache pollution. As a result, Leap provides Memcached with almost the local memory level behavior at 50% memory limit while the default data path of Infiniswap faces 10.1% throughput loss (Figure 2.9d). At 25% memory limit, Leap deviates from the local memory throughput behavior by only 1.7%. Here, the default data path of Infiniswap faces 18.49% throughput loss. In this phase, Leap improves Infiniswap's throughput by $1.11\times$ and $1.21\times$ at 50% and 25% memory limits, respectively. Here, Leap provides with $5.94\times$ and $1.08\times$ better 99^{th} percentile 4KB page access time at 50% and 25% cases, respectively.

Performance Under Constrained Cache Size To observe Leap's performance benefit in the presence of limited cache size, we run the four applications in 50% memory limit configuration at different cache limits (Figure 2.10).

For Memcached, as most of the accesses are of random patterns, most of the performance benefit comes from Leap's faster slow path. For the rest of the applications, as the prefetcher has better timeliness, most of the prefetched caches get used and evicted before the cache size hits the limit. Hence, during O(1) MB cache size, all of these applications face minimal performance drop (11.87–13.05%) compared to the unlimited cache space scenario. Note that, for NumPy, 3.2 MB cache size is only 0.02% of its total remote memory usage.

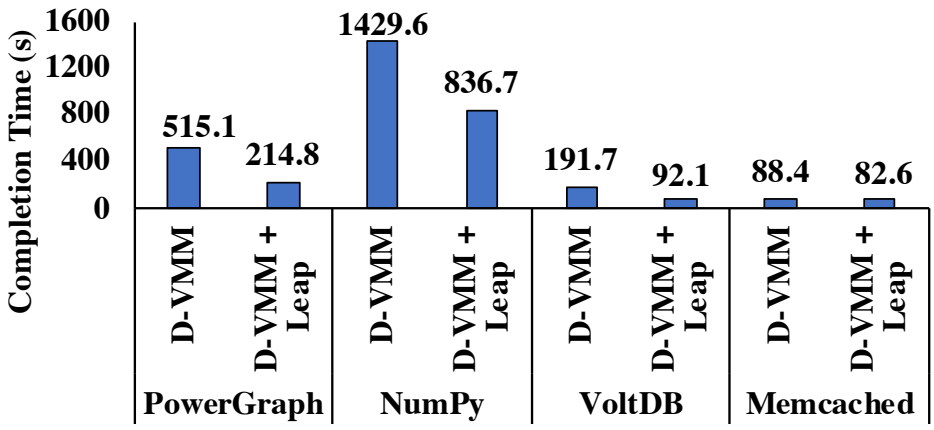


Figure 2.11: Leap improves application-level performance when all four applications access remote memory concurrently.

Multiple Applications Running Together We run all four applications on a single host machine simultaneously with their 50% memory limit and observe the performance benefit of Leap for Infiniswap when multiple throughput- (PowerGraph, NumPy) and latency-sensitive applications (VoltDB, Memcached) concurrently request for remote memory access (Figure 2.11). As Leap isolates each application's page access path, its prefetcher can consider individual access patterns while making prefetch decisions. Therefore, it brings more accurate remote pages for each of the applications and reduces the contention over the network. As a result, overall application-level performance improves by 1.1–2.4× over Infiniswap. To enable aggregate performance comparison, here, we present the end-to-end completion time of application-workload combinations defined earlier; application-specific metrics improve as well.

2.6 Discussion and Future Work

Thread-specific Prefetching Linux kernels today manage memory address space at the process level. Thread-specific page access tracking requires a significant change in the whole virtual memory subsystem. However, this would help efficiently identify multiple concurrent streams from different threads. Low-overhead, thread-specific page access tracking and prefetching can be an interesting research direction.

Concurrent Disk and Remote I/O Leap's prefetcher can be used for both disaggregated and existing Linux Kernels. Currently, Leap runs as a single memory management module on the host server where paging is allowed through either existing block layers or Leap's remote memory data path. The current implementation does not allow the concurrent use of both block layer and remote

memory. Exploring this direction can lead to further benefits for systems using Leap.

Optimized Remote I/O Interface In this work, we focused on augmenting existing memory disaggregation frameworks with a leaner and efficient data path. This allowed us to keep Leap transparent to the remote I/O interface. We believe that exploring the effects of load balancing, fault-tolerance, data locality, and application-specific isolation in remote memory as well as an optimized remote I/O interface are all potential future research directions.

2.7 Related Work

Remote Memory Solutions A large number of software systems have been proposed over the years to access remote machine's memory for paging [151, 223, 241, 120, 145, 211, 3, 270, 165, 275, 203], global virtual machine abstraction [150, 75, 198], and distributed data stores and file systems [248, 142, 212, 197, 92]. Hardware-based remote access using PCIe interconnects [215] or extended NUMA memory fabric [245] are also proposed to disaggregate memory. Leap is complementary to these works.

Kernel Data Path Optimizations With the emergence of faster storage devices, several optimization techniques, and design principles have been proposed to fully utilize faster hardware. Considering the overhead of the block layer, different service level optimizations and system redesigns have been proposed – examples include parallelism in batching and queuing mechanism [110, 309], avoiding interrupts and context switching during I/O scheduling [313, 307, 117, 95], better buffer cache management [175], etc. During remote memory access, optimization in data path has been proposed through request batching [180, 288, 179], eliminating page migration bottleneck [306], reducing remote I/O bandwidth through compression [203], and network-level block devices [211]. Leap's data path optimizations are inspired by many of them.

Prefetching Algorithms Many prefetching techniques exist to utilize hardware features [173, 177, 279, 330], compiler-injected instructions [253, 188, 256, 152, 194], and memory-side access pattern [240, 283, 280, 148, 281] for cache line prefetching. They are often limited to specific access patterns, application behavior, or require specified hardware design. More importantly, they are designed for a lower level memory stack.

A large number of entirely kernel-based prefetching techniques have also been proposed to hide the latency overhead of file accesses and page faults [184, 148, 164, 113, 302]. Among them, Linux Read-Ahead [302] is the most widely used. However, it does not consider the access history

to make prefetch decisions. It was also designed for hiding disk seek time. Therefore, its optimistic looking around approach often results in lower cache utilization for remote memory access.

To the best of our knowledge, Leap is the first to consider a fully software-based, kernel-level prefetching technique for DRAM with remote memory as a backing storage over fast RDMA-capable networks.

2.8 Conclusion

The study presents Leap, a remote page prefetching algorithm that relies on majority-based pattern detection instead of strict detection. As a result, Leap is resilient to short-term irregularities in page access patterns of multi-threaded applications. We implement Leap in a leaner and faster data path in the Linux kernel for remote memory access over RDMA without any application or hardware modifications.

Our integrations of Leap with two major memory disaggregation systems (namely, Infiniswap and Remote Regions) show that the median and tail remote page access latencies improves by up to $104.04\times$ and $22.62\times$, respectively, over the state-of-the-art. This, in turn, leads to application-level performance improvements of 1.27–$10.16\times$. Finally, Leap's benefits extend beyond disaggregated memory – applying it to HDD and SSD leads to considerable performance benefits as well.

Leap is available at https://github.com/SymbioticLab/leap.

CHAPTER 3

Hydra : Resilient and Highly Available Remote Memory

3.1 Introduction

Modern datacenters are embracing a paradigm shift toward disaggregation, where each resource is decoupled and connected through a high-speed network fabric [49, 35, 61, 275, 203, 134, 135, 215, 213, 44]. In such disaggregated datacenters, each server node is specialized for specific purposes – some are specialized for computing, while others for memory, storage, and so on. Memory, being the prime resource for high-performance services, is becoming an attractive target for disaggregation [215, 93, 165, 92, 157, 203, 131, 265, 111].

Recent remote-memory frameworks allow an unmodified application to access remote memory in an implicit manner via well-known abstractions such as distributed virtual file system (VFS) and distributed virtual memory manager (VMM) [92, 165, 157, 224, 275, 203, 321]. With the advent of RDMA, remote-memory solutions are now close to meeting the single-digit μs latency required to support acceptable application-level performance [157, 203]. However, realizing remote memory for heterogeneous workloads running in a large-scale cluster faces considerable challenges [93, 115] stemming from two root causes:

1. *Expanded failure domains:* As applications rely on memory across multiple machines in a remote-memory cluster, they become susceptible to a wide variety of failure scenarios. Potential failures include independent and correlated failures of remote machines, evictions from and corruptions of remote memory, and network partitions.

2. *Tail at scale:* Applications also suffer from stragglers or late-arriving remote responses. Stragglers can arise from many sources including latency variabilities in a large network due to congestion and background traffic [139].

While one leads to catastrophic failures and the other manifests as service-level objective (SLO) violations, both are unacceptable in production [203, 235]. Existing solutions take three primary approaches to address them: (i) local disk backup [165, 275], (ii) remote in-memory replication [223, 153, 180, 142], and (iii) remote in-memory erasure coding [271, 259, 310, 119] and compression [203]. Unfortunately, they suffer from some combinations of the following problems.

High latency: Disk backup has no additional memory overhead, but the access latency is intolerably high under any correlated failures. Systems that take the third approach do not meet the single-digit μs latency requirement of remote memory even when paired with RDMA (Figure 3.1).

High cost: Replication has low latency, but it doubles memory consumption and network bandwidth requirements. Disk backup and replication represent the two extreme points in the performance-vs-efficiency tradeoff space (Figure 3.1).

Low availability: All three approaches lose availability to low latency memory when even a very small number of servers become unavailable. With the first approach, if a single server fails its data needs to be reconstituted from disk, which is a slow process. In the second and third approach, when even a small number of servers (e.g., three) fail simultaneously, some users will lose access to data. This is due to the fact that replication and erasure coding assign replicas and coding groups to *random* servers. Random data placement is susceptible to data loss when a small number of servers fail at the same time [128, 127] (Figure 3.2).

In this study, we consider how to mitigate these problems and present Hydra, a low-latency, low-overhead, and highly available resilience mechanism for remote memory. While erasure codes are known for reducing storage overhead and for better load balancing, it is challenging for remote memory with μs-scale access requirements (preferably, 3-5μs) [157]. We demonstrate how to achieve resilient erasure-coded cluster memory with single-digit μs latency even under simultaneous failures at a reduced data amplification overhead.

We explore the challenges and tradeoffs for resilient remote memory without sacrificing application-level performance or incurring high overhead in the presence of correlated failures (§4.2). We also explore the trade-off between load balancing and high availability in the presence of simultaneous server failures. Our solution, Hydra, is a configurable resilience mechanism that applies online erasure coding to individual remote memory pages while maintaining high availability (§3.3). Hydra's carefully designed data path enables it to access remote memory pages within a single-digit μs median and tail latency (§3.4). Furthermore, we develop CodingSets, a novel coding group placement algorithm for erasure codes that provides load balancing while reducing the probability of data loss under correlated failures (§3.5).

We develop Hydra as a drop-in resilience mechanism that can be applied to existing remote memory frameworks [92, 165, 224, 275, 111]. We integrate Hydra with the two major remote memory approaches widely embraced today: disaggregated VMM (used by Infiniswap [165], and

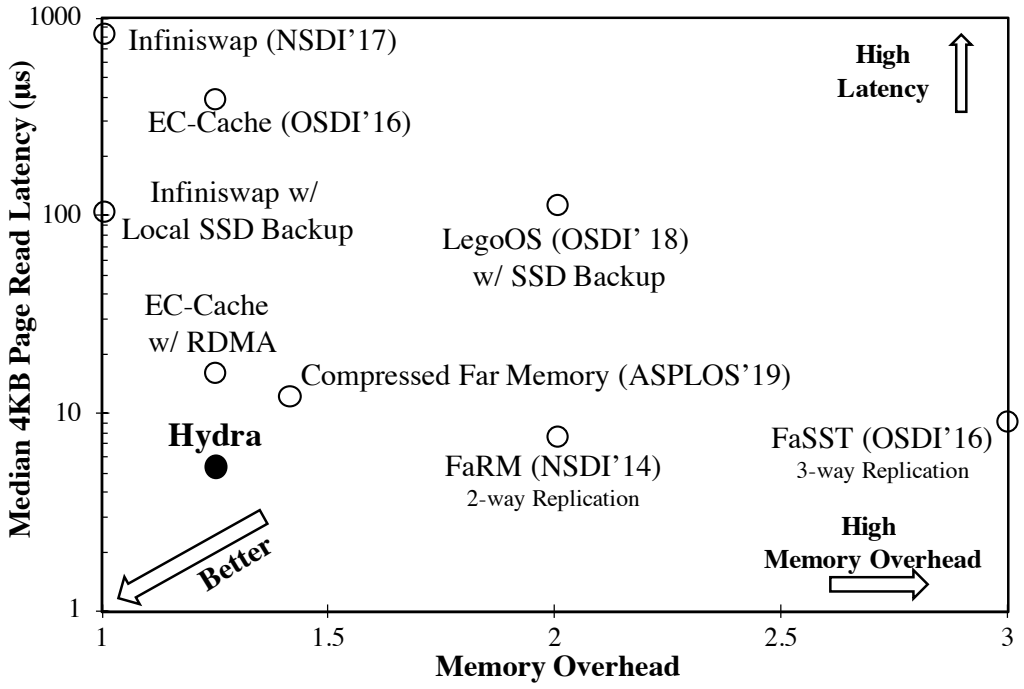


Figure 3.1: Performance-vs-efficiency tradeoff in the resilient cluster memory design space. Here, the Y-axis is in log scale.

Leap [224]) and disaggregated VFS (used by Remote Regions [92]) (§3.6). Our evaluation using production workloads shows that Hydra achieves the best of both worlds (§5.6). Hydra closely matches the performance of replication-based resilience with 1.6× lower memory overhead with or without the presence of failures. At the same time, it improves latency and throughput of the benchmark applications by up to 64.78× and 20.61×, respectively, over SSD backup-based resilience with only 1.25× higher memory overhead. While providing resiliency, Hydra also improves the application-level performance by up to $4.35\times$ over its counterparts. CodingSets reduces the probability of data loss under simultaneous server failures by about $10\times$. Hydra is available at https://github.com/SymbioticLab/hydra.

Contributions In this study, we make the following contributions:

- Hydra is the first in-memory erasure coding scheme that achieves single-digit μs tail memory access latency.
- Novel analysis of load balancing and availability trade-off for distributed erasure codes.
- CodingSets is a new data placement scheme that balances availability and load balancing, while reducing probability of data loss by an order of magnitude during failures.

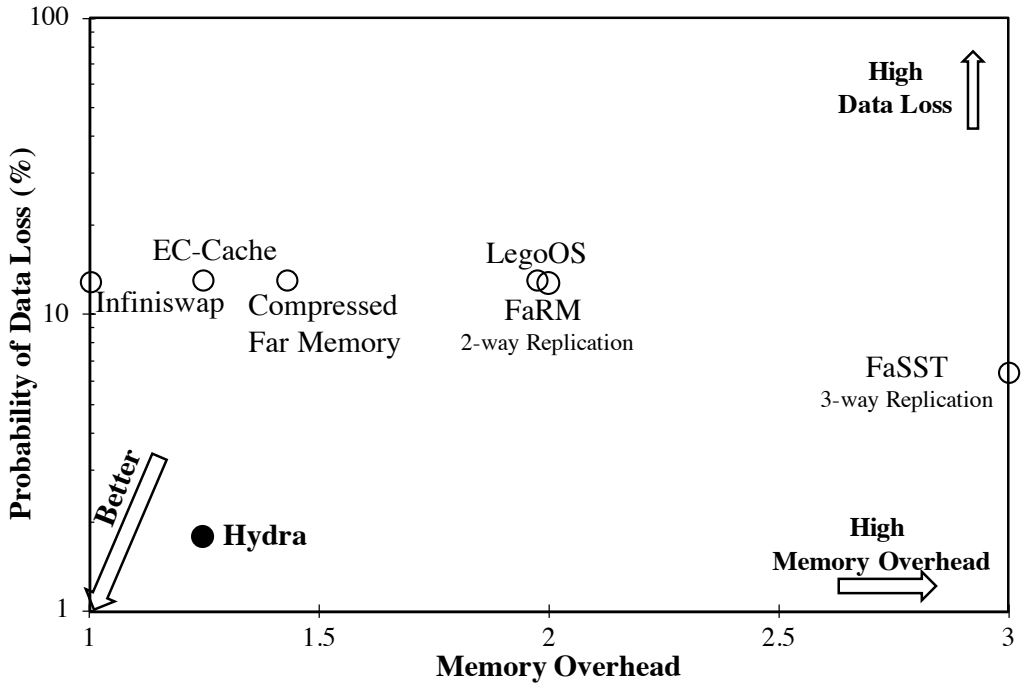


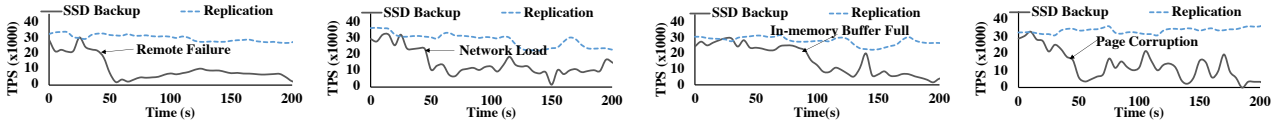
Figure 3.2: Availability-vs-efficiency tradeoff considering 1% simultaneous server failures in a 1000-machine cluster.

3.2 Background and Motivation

3.2.1 Remote Memory

Remote memory exposes memory available in remote machines as a pool of memory shared by many machines. It is often implemented logically by leveraging stranded memory in remote machines via well-known abstractions, such as the file abstraction [92], remote memory paging [165, 157, 211, 224, 111], and virtual memory management for distributed OS [275]. In the past, specialized memory appliances for physical memory disaggregation were proposed as well [215, 216].

All existing remote-memory solutions use the 4KB page granularity. While some applications use huge pages for performance enhancement [201], the Linux kernel still performs paging at the basic 4KB level by splitting individual huge pages because huge pages can result in high amplification for dirty data tracking [112]. Existing remote-memory systems use disk backup [165, 275] and in-memory replication [223, 153] to provide availability during failures.



(a) Remote failure (b) Background Load (c) Request burst (d) Page Corruption

Figure 3.3: TPC-C throughput over time on VoltDB when 50% of the working set fits in memory. Arrows point to uncertainty injection time.

3.2.2 Failures in Remote Memory

The probability of failure or temporary unavailability is higher in a large remote-memory cluster, since memory is being accessed remotely. To illustrate possible performance penalties in the presence of such unpredictable events, we consider a resilience solution from the existing literature [165], where each page is asynchronously backed up to a local SSD. We run TPC-C [79] on an in-memory database system, VoltDB [84]. We set VoltDB's available memory to 50% of its peak memory to force remote paging for up to 50% of its working set.

1. Remote Failures and Evictions Machine failures are the norm in large-scale clusters where thousands of machines crash over a year due to a variety of reasons, including software and hardware failures [319, 136, 132, 154]. Concurrent failures within a rack or network segments are quite common and typically occur dozens of times a year. Even cluster-wide power outage is not uncommon – occurs once or twice per year in a datacenter. For example, during a recent cluster-wide power outage in Google Cloud, around 23% of the machines were unavailable for hours [40].

Without redundancy, applications relying on remote memory may fail when a remote machine fails or remote memory pages are evicted. As disk operations are significantly slower than the latency requirement of remote memory, disk-based fault-tolerance is far from being practical. In the presence of a remote failure, VoltDB experiences almost 90% throughput loss (Figure 3.3a); throughput recovery takes a long time after the failure happens.

2. Background Network Load Network load throughout a large cluster can experience significant fluctuations [182, 139], which can inflate RDMA latency and application-level stragglers, causing unpredictable performance issues [323, 324]. In the presence of an induced bandwidth-intensive background load, VoltDB throughput drops by about 50% (Figure 3.3b).

3. Request Bursts Applications can have bursty memory access patterns. Existing solutions maintain an in-memory buffer to absorb temporary bursts [250, 92, 165]. However, as the buffer ties remote access latency to disk latency when it is full, the buffer can become the bottleneck when

a workload experiences a prolonged burst. While a page read from remote memory is still fast, backup page writes to the local disk become the bottleneck after the 100^{th} second in Figure 3.3c. As a result, throughput drops by about 60%.

4. Memory Corruption During remote memory access, if any one of the remote servers experiences a corruption, or if the memory gets corrupted over the network a memory corruption event will occur. In such case, disk access causes failure-like performance loss (Figure 3.3d).

Performance vs. Efficiency Tradeoff for Resilience In all of these scenarios, the obvious alternative – in-memory $2\times$ or $3\times$ replication [223, 153] – is effective in mitigating a small-scale failure, such as the loss of a single server (Figure 3.3a). When an in-memory copy becomes unavailable, we can switch to an alternative. Unfortunately, replication incurs high memory overhead in proportion to the number of replicas. This defeats the purpose of remote memory. Hedging requests to avoid stragglers [139] in a replicated system doubles its bandwidth requirement.

This leads to an impasse: one has to either settle for high latency in the presence of a failure or incur high memory overhead. Figure 3.1 depicts this performance-vs-efficiency tradeoff under failures and memory usage overhead to provide resilience. Beyond the two extremes in the tradeoff space, there are two primary alternatives to achieve high resilience with low overhead. The first is replicating pages to remote memory after compressing them (e.g., using zswap) [203], which improves the tradeoff in both dimensions. However, its latency can be more than 10μs when data is in remote memory. Especially, during resource scarcity, the presence of a prolonged burst in accessing remote compressed pages can even lead to orders of magnitude higher latency due to the demand spike in both CPU and local DRAM consumption for decompression. Besides, this approach faces similar issues as replication such as latency inflation due to stragglers.

The alternative is erasure coding, which has recently made its way from disk-based storage to in-memory cluster caching to reduce storage overhead and improve load balancing [91, 118, 300, 119, 310, 259]. Typically, an object is divided into k *data splits* and encoded to create r equal-sized *parity splits* ($k > r$), which are then distributed across $(k + r)$ failure domains. Existing erasure-coded memory solutions deal with large objects (e.g., larger than 1 MB [259]), where hundreds-of-μs latency of the TCP/IP stack can be ignored. Simply replacing TCP with RDMA is not enough either. For example, the EC-Cache with RDMA (Figure 3.1) provides a lower storage overhead than compression but with a latency around 20μs.

Last but not least, all of these approaches experience high unavailability in the presence of correlated failures [128].

3.2.3 Challenges in Erasure-Coded Memory

High Latency Individually erasure coding 4 KB pages that are already small lead to even smaller data chunks ($\frac{4}{k}$ KB), which contributes to the higher latency of erasure-coded remote memory over RDMA due to following primary reasons:

1. ***Non-negligible coding overhead:*** When using erasure codes with on-disk data or over slower networks that have hundreds-of-μs latency, its 0.7μs encoding and 1.5μs decoding overheads can be ignored. However, they become non-negligible when dealing with DRAM and RDMA.

2. ***Stragglers and errors:*** As erasure codes require k splits before the original data can be constructed, any straggler can slow down a remote read. To detect and correct an error, erasure codes require additional splits; an extra read adds another round-trip to double the overall read latency.

3. ***Interruption overhead:*** Splitting data also increases the total number of RDMA operations for each request. Any context switch in between can further add to the latency.

4. ***Data copy overhead:*** In a latency-sensitive system, additional data movement can limit the lowest possible latency. During erasure coding, additional data copy into different buffers for data and parity splits can quickly add up.

Availability Under Simultaneous Failures Existing erasure coding schemes can handle a small-scale failure without interruptions. However, when a relatively modest number of servers fail or become unavailable at the same time (e.g., due to a network partition or a correlated failure event), they are highly susceptible to losing availability to some of the data.

This is due to the fact that existing erasure coding schemes generate coding groups on random sets of servers [259]. In a coding scheme with k data and r parity splits, an individual coding group, will fail to decode the data if $r + 1$ servers fail simultaneously. Now in a large cluster with $r + 1$ failures, the probability that those $r + 1$ servers will fail for a *specific* coding group is low. However, when coding groups are generated randomly (i.e., each one of them compromises a random set of $k + r$ servers), and there are a large number of coding groups per server, then the probability that those $r + 1$ servers will affect *any* coding group in the cluster is much higher. Therefore, state-of-the-art erasure coding schemes, such as EC-Cache, will experience a very high probability of unavailability even when a very small number of servers fail simultaneously.

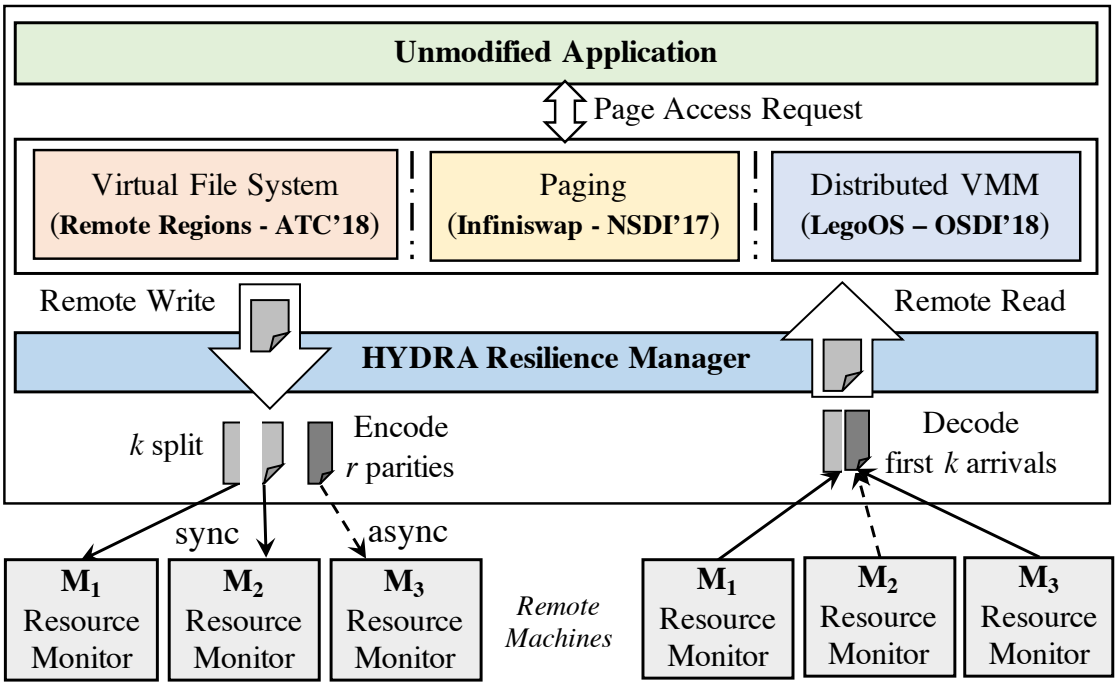


Figure 3.4: Resilience Manager provides with resilient, erasure-coded remote memory abstraction. Resource Monitor manages the remote memory pool. Both can be present in a machine.

3.3 Hydra Architecture

Hydra is an erasure-coded resilience mechanism for existing remote-memory techniques to provide better performance-efficiency tradeoff under remote failures while ensuring high availability under simultaneous failures. It has two main components (Figure 3.4): (i) **Resilience Manager** coordinates erasure-coded resilience operations during remote read/write; (ii) **Resource Monitor** handles the memory management in a remote machine. Both can be present in every machine and work together without central coordination.

3.3.1 Resilience Manager

Hydra Resilience Manager provides remote memory abstraction to a client machine. When an unmodified application accesses remote memory through different state-of-the-art remote-memory solutions (e.g., via VFS or VMM), the Resilience Manager transparently handles all aspects of RDMA communication and erasure coding. Each client has its own Resilience Manager that handles slab placement through CodingSets, maintains remote slab-address mapping, performs erasure-coded RDMA read/write. Resilience Manager communicates to Resource Monitor(s) running on remote memory host machines, performs remote data placement, and ensures resilience. As a client's Resilience Manager is responsible for the resiliency of its remote data, the Resilience

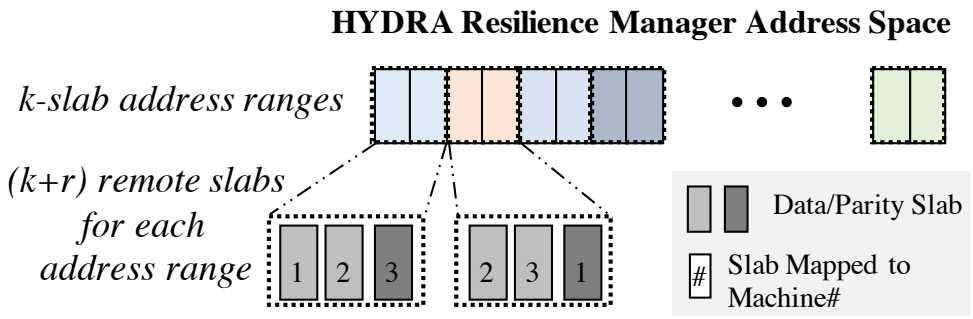


Figure 3.5: Hydra's address space is divided into fixed-size address ranges, each of which spans $(k + r)$ memory slabs in remote machines; i.e., k for data and r for parity (here, k=2 and r=1).

Managers do not need to coordinate with each other.

Following the typical (k, r) erasure coding construction, the Resilience Manager divides its remote address space into fixed-size *address ranges*. Each address range resides in $(k + r)$ remote *slabs*: k slabs for page data and r slabs for parity (Figure 3.5). Each of the $(k + r)$ slabs of an address range are distributed across $(k + r)$ independent failure domains using CodingSets (§3.5). Page accesses are directed to the designated $(k+r)$ machines according to the address–slab mapping. Although remote I/O happens at the page level, the Resilience Manager coordinates with remote Resource Monitors to manage coarse-grained memory slabs to reduce metadata overhead and connection management complexity.

3.3.2 Resource Monitor

Resource Monitor manages a machine's local memory and exposes them to the remote Resilience Manager in terms of fixed-size ($SlabSize$) memory slabs. Different slabs can belong to different machines' Resilience Manager. During each control period ($ControlPeriod$), the Resource Monitor tracks the available memory in its local machine and proactively allocates (reclaims) slabs to (from) remote mapping when memory usage is low (high). It also performs slab regeneration during remote failures or corruptions.

Fragmentation in Remote Memory During the registration of Resource Monitor(s), Resilience Manager registers the RDMA memory regions and allocates slabs on the remote machines based on its memory demand. Memory regions are usually large (by default, 1GB) and the whole address space is homogeneously splitted. Moreover, RDMA drivers guarantee the memory regions are generated in a contiguous physical address space to ensure faster remote-memory access. Hydra introduces no additional fragmentation in remote machines.

3.3.3 Failure Model

Assumptions In a large remote-memory cluster, (a) remote servers may crash or networks may become partitioned; (b) remote servers may experience memory corruption; (c) the network may become congested due to background traffic; and (d) workloads may have bursty access patterns. These events can lead to catastrophic application-failures, high tail latencies, or unpredictable performance. Hydra addresses all of these uncertainties in its failure domain. Although Hydra withstands a remote-network partition, as there is no local-disk backup, it cannot handle local-network failure. In such cases, the application is anyways inaccessible.

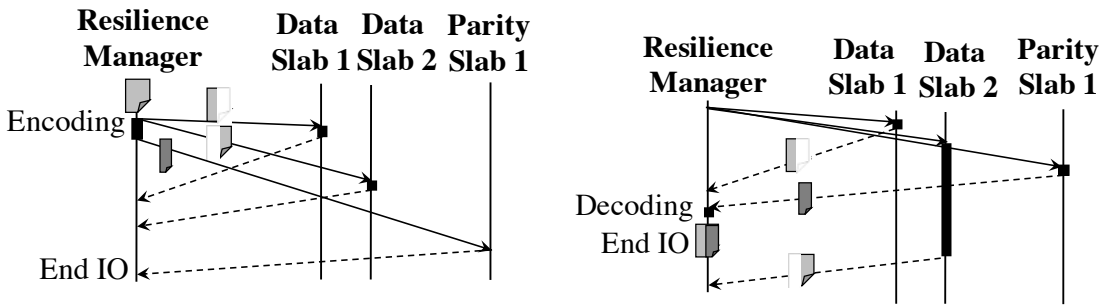
Single vs. Simultaneous Failure A single node failure means the unavailability of slabs in a remote machine. In such an event, all the data or parity allocated on the slab(s) become unavailable. As we spread the data and parity splits for a page across multiple remote machines (§3.5), during a single node failure, we assume that only a single data or parity split for that page is being affected.

Simultaneous host failures typically occur due to large-scale failures, such as power or network outage that cause multiple machines to become unreachable. In such a case, we assume multiple data and/or parity splits for a page become unavailable. Note that in both cases, the data is unavailable, but not compromised. Resilience Manager can detect the unreachability and communicate to other available Resource Monitor(s) on to regenerate specific slab(s).

3.4 Resilient Data Path

Hydra can operate on different resilient modes based on a client's need – **(a)** *Failure Recovery*: provides resiliency in the presence of any remote failure or eviction; **(b)** *Corruption Detection*: only detects the presence of corruption in remote memory; **(c)** *Corruption Correction*: detects and corrects remote memory corruption; and **(d)** *EC-only mode*: provides erasure-coded faster remote-memory data path without any resiliency guarantee. Note that both of the corruption modes by default inherit the *Failure Recovery* mode.

Before initiating the Resilience Manager, one needs to configure Hydra to a specific mode according to the resilience requirements and memory overhead concerns (Table 3.1). Multiple resilience modes cannot act simultaneously, and the modes do not switch dynamically during runtime. In this section, we present Hydra's data path design to address the resilience challenges mentioned in §4.2.3.



(a) Remote Write (b) Remote Read

Figure 3.6: To handle failures, Hydra (a) first writes data splits, then encodes/writes parities to hide encoding latency; (b) reads from $k + \Delta$ slabs to avoid stragglers, finishes with first k arrivals.

3.4.1 Hydra Remote Memory Data Path

To minimize erasure coding's latency overheads, Resilience Manager's data path incorporate following design principles.

3.4.1.1 Asynchronously Encoded Write

To hide the erasure coding latency, existing systems usually perform batch coding where multiple pages are encoded together. The encoder waits until a certain number of pages are available. This idle waiting time can be insignificant compared to disk or slow network (e.g., TCP) access. However, to maintain the tail latency of a remote I/O within the single-digit μs range, this "batch waiting" time needs to be avoided.

During a remote write, Resilience Manager applies erasure coding within each individual page by dividing it into k splits (for a 4 KB page, each split size is $\frac{4}{k}$KB), encodes these splits using Reed-Solomon (RS) codes [262] to generate r parity splits. Then, it writes these $(k + r)$ splits to different $(k + r)$ slabs that have already been mapped to unique remote machines. Each Resilience Manager can have their own choice of k and r. This individual page-based coding decreases latency by avoiding the "batch waiting" time. Moreover, the Resilience Manager does not have to read unnecessary pages within the same batch during remote reads, which reduces bandwidth overhead. Distributing remote I/O across many remote machines increases I/O parallelism too.

Resilience Manager sends the data splits first, then it encodes and sends the parity splits asynchronously. Decoupling the two hides encoding latency and subsequent write latency for the parities without affecting the resilience guarantee. In the absence of failure, any k successful writes of the $(k + r)$ allow the page to be recovered. However, to ensure resilience guarantee for r failures, all $(k + r)$ must be written. In the *failure recovery* mode, a write is considered complete after all $(k + r)$ have been written. In the *corruption correction (detection)* mode, to correct (detect) Δ

Resilience Mode	# of Errors	Minimum # of Splits	Memory Overhead
Failure Recovery	r	k	$1 + \frac{r}{k}$
Corruption Detection	Δ	$k + \Delta$	$1 + \frac{\Delta}{k}$
Corruption Correction	Δ	$k + 2\Delta + 1$	$1 + \frac{2\Delta+1}{k}$
EC-only	–	k	$1 + \frac{r}{k}$

Table 3.1: Minimum number of splits needs to be written to/read from remote machines for resilience during a remote I/O.

corruptions, a write waits for $k + 2\Delta + 1$ $(k + \Delta)$ to be written. If the acknowledgement fails to reach the Resilience Manager due to a failure in the remote machine, the write for that split is considered failed. Resilience Manager tries to write that specific split(s) after a timeout period to another remote machine. Figure 3.6a depicts the timeline of a page write in the *failure recovery mode*.

3.4.1.2 Late-Binding Resilient Read

During read, any k out of the $k+r$ splits suffice to reconstruct a page. However, in *failure recovery* mode, to be resilient in the presence of Δ failures, during a remote read, Hydra Resilience Manager reads from $k+\Delta$ randomly chosen splits in parallel. A page can be decoded as soon as any k splits arrive out of $k+\Delta$. The additional Δ reads mitigate the impact of stragglers on tail latency as well. Figure 3.6b provides an example of a read operation in the *failure recovery* mode with $k = 2$ and $\Delta = 1$, where one of the data slabs (Data Slab 2) is a straggler. $\Delta = 1$ is often enough in practice.

If simply "detect and discard corrupted memory" is enough for any application, one can configure Hydra with *corruption detection* mode and avoid the extra memory overhead of *corruption correction* mode. In *corruption detection* mode, before decoding a page, the Resilience Manager waits for $k + \Delta$ splits to arrive to *detect* Δ corruptions. After a certain amount of corruptions, Resilience Manager marks the machine(s) with corrupted splits as probable erroneous machines, initiates a background slab recovery operation, and avoids them during future remote I/O.

To correct the error, in *corruption correction* mode, when an error is detected, it requests additional $\Delta + 1$ reads from the rest of the $k + r$ machines. Otherwise, the read completes just after the arrival of the $k + \Delta$ splits. If the error rate for a remote machine exceeds a user-defined threshold ($ErrorCorrectionLimit$), subsequent read requests involved with that machine initiates with $k + 2\Delta + 1$ split requests as there is a high probability to reach an erroneous machine. This will reduce the wait time for additional $\Delta + 1$ reads. This continues until the error rate for the involved machine gets lower than the $ErrorCorrectionLimit$. If this continues for long and/or the error rate goes beyond another threshold ($SlabRegenerationLimit$), Resilience Manager initiates a slab regeneration request for that machine.

One can configure Hydra with *EC-only* mode to access erasure-coded remote memory and benefit from the fast data path without any resiliency guarantee. In this mode, a remote I/O completes just after writing/reading any k splits. Table 3.1 summarizes the minimum number of splits the Resilience Manager requires to write/read during a remote I/O operation to provide resiliency in different modes.

Overhead of Replication To remain operational after r failures, in-memory replication requires at least $r+1$ copies of an entire 4 KB page, and hence the memory overhead is $(r+1)\times$. However, a remote I/O operation can complete just after the confirmation from one of the $r+1$ machines. To detect and fix Δ corruptions, replication needs $\Delta+1$ and $2\Delta+1$ copies of the *entire* page, respectively. Thus, to provide the correctness guarantee over Δ corruptions, replication needs to wait until it writes to or reads from at least $2\Delta+1$ of the replicas along with a memory overhead of $(2\Delta+1)\times$.

3.4.1.3 Run-to-Completion

As Resilience Manager divides a 4 KB page into k smaller pieces, RDMA messages become smaller. In fact, their network latency decrease to the point that run-to-completion becomes more beneficial than a context switch. Hence, to avoid interruption-related overheads, the remote I/O request thread waits until the RDMA operations are done.

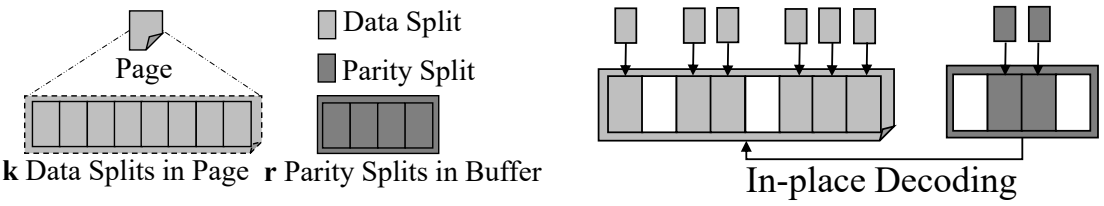
3.4.1.4 In-Place Coding

To reduce the number of data copies, Hydra Resilience Manager uses in-place coding with an extra buffer of r splits. During a write, the data splits are always kept in-page while the encoded r parities are put into the buffer (Figure 3.7a). Likewise, during a read, the data splits arrive at the page address, and the parity splits find their way into the buffer (Figure 3.7b).

In the failure recovery mode, a read can complete as soon as any k valid splits arrive. Corrupted/straggler data split(s) can arrive late and overwrite valid page data. To address this, as soon as Hydra detects the arrival of k valid splits, it deregisters relevant RDMA memory regions. It then performs decoding and directly places the decoded data in the page destination. Because the memory region has already been deregistered, any late data split cannot access the page. During all remote I/O, requests are forwarded directly to RDMA dispatch queues without additional copying.

3.4.2 Handling Uncertainties

Remote Failure Hydra uses reliable connections (RC) for all RDMA communication. Hence, we consider unreachability due to machine failures/reboots or network partition as the primary



(a) Remote Write (b) Remote Read

Figure 3.7: Hydra performs in-place coding with a buffer of r splits to reduce the data-copy latency.

cause of failure. When a remote machine becomes unreachable, the Resilience Manager is notified by the RDMA connection manager. Upon disconnection, it processes all the in-flight requests in order first. For ongoing I/O operations, it resends the I/O request to other available machines. Since RDMA guarantees strict ordering, in the read-after-write case, read requests will arrive at the same RDMA dispatch queue after write requests; hence, read requests will not be served with stale data. Finally, Hydra marks the failed slabs and future requests are directed to the available ones. If the Resource Monitor in the failed machine revives and communicates later, Hydra reconsiders the machine for further remote I/O.

Adaptive Slab Allocation/Eviction Resource Monitor allocates memory slabs for Resilience Managers as well as proactively frees/evicts them to avoid local performance impacts (Figure 3.8). It periodically monitors local memory usage and maintains a headroom to provide enough memory for local applications. When the amount of free memory shrinks below the headroom (Figure 3.8a), the Resource Monitor first proactively frees/evicts slabs to ensure local applications are unaffected. To find the eviction candidates, we avoid random selection as it has a higher likelihood of evicting a busy slab. Rather, we uses the decentralized batch eviction algorithm [165] to select the least active slabs. To evict E slabs, we contact $(E + E')$ slabs (where $E' \leq E$) and find the least-frequently-accessed slabs among them. This doesn't require to maintain a global knowledge or search across all the slabs.

When the amount of free memory grows above the headroom (Figure 3.8b), the Resource Monitor first attempts to make the local Resilience Manager to reclaim its pages from remote memory and unmap corresponding remote slabs. Furthermore, it proactively allocates new, unmapped slabs that can be readily mapped and used by remote Resilience Managers.

Background Slab Regeneration The Resource Monitor also regenerates unavailable slabs – marked by the Resilience Manager – in the background. During regeneration, writes to the slab are disabled to prevent overwriting new pages with stale ones; reads can still be served without

interruption.

Hydra Resilience Manager uses the placement algorithm to find a new regeneration slab in a remote Resource Monitor with a lower memory usage. It then hands over the task of slab regeneration to that Resource Monitor. The selected Resource Monitor decodes the unavailable slab by directly reading the k randomly-selected remaining valid slab for that address region. Once regeneration completes, it contacts the Resilience Manager to mark the slab as available. Requests thereafter go to the regenerated slab.

3.5 CodingSets for High Availability

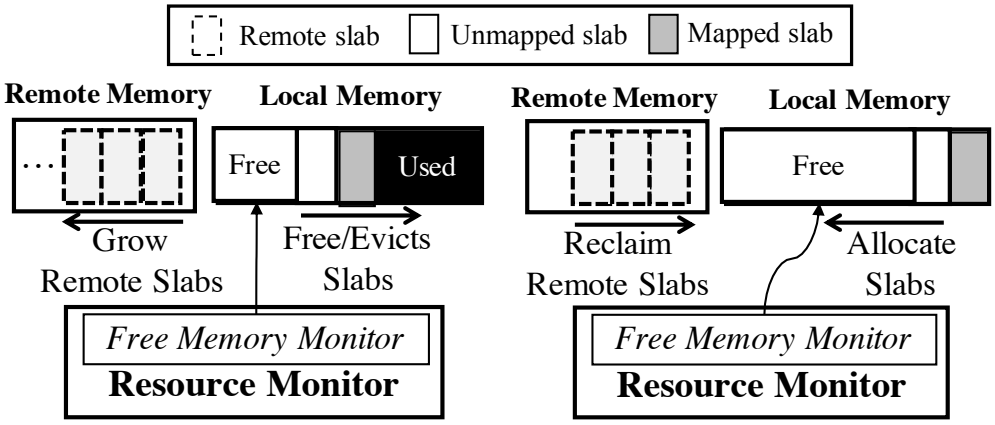
Hydra uses CodingSets, a novel coding group placement scheme to perform load-balancing while reducing the probability of data loss. Prior works show orders-of-magnitude more frequent data loss due to events causing multiple nodes to fail simultaneously than data loss due to independent node failures [154, 127]. Several scenarios can cause multiple servers to fail or become unavailable simultaneously, such as network partitions, partial power outages, and software bugs. For example, a power outage can cause 0.5%-1% machines to fail or go offline concurrently [128]. In case of Hydra, data loss will happen if a concurrent failure kills more than $r+1$ of $(k+r)$ machines for a particular *coding group*.

We are inspired by copysets, a scheme for preventing data loss under correlated failures in replication [128, 127], which constraints the number of replication groups, in order to *reduce the frequency of data loss events*. Using the same terminology as prior work, we define each unique set of $(k+r)$ servers within a coding group as a $copyset$. The number of copysets in a single coding group will be: $\binom{k+r}{r+1}$. For example, in an (8+2) configuration, where nodes are numbered $1, 2, \ldots, 10$, the 3 nodes that will cause failure if they fail at the same time (i.e., copysets) will be every 3 combinations of 10 nodes: $(1,2,3), (1,2,4), \ldots, (8,9,10)$, and the total number of copysets will be $\binom{10}{3} = 120$.

For a data loss event impacting exactly $r+1$ random nodes simultaneously, the probability of losing data of a single specific coding group: $\mathbb{P}[Group] = \frac{\text{Num. of Copysets in Coding Group}}{\text{Total Copysets}} = \frac{\binom{k+r}{r+1}}{\binom{N}{r+1}}$, where N is the total number of servers.

In a cluster with more than $(k+r)$ servers, we need to use more than one coding group. However, if each server is a member of a single coding group, hot spots can occur if one or more members of that group are overloaded. Therefore, for load-balancing purposes, a simple solution is to allow each server to be a member of multiple coding groups, in case some members of a particular coding group are over-loaded at the time of online coding.

Assuming we have G *disjoint* coding groups, and the correlated failure rate is $f\%$, the total probability of data loss is: $1 - (1 - \mathbb{P}[Group] \cdot G)^{\binom{N \cdot f}{r+1}}$. We define disjoint coding groups where



(a) High memory pressure (b) Low memory pressure

Figure 3.8: Resource Monitor proactively allocates memory for remote machines and frees local memory pressure.

the groups do not share any copysets; or in other words, they do not overlap by more than r nodes.

Strawman: Multiple Coding Groups per Server In order to equalize load, we consider a scheme where each slab forms a coding group with the least-loaded nodes in the cluster at coding time. We assume the nodes that are least loaded at a given time are distributed randomly, and the number of slabs per server is S. When $S \cdot (r+k) \ll N$, the coding groups are highly likely to be disjoint [128], and the number of groups is equal to: $G = \frac{N \cdot S}{k+r}$.

We call this placement strategy the *EC-Cache scheme*, as it produces a random coding group placement used by the prior state-of-the-art in-memory erasure coding system, EC-Cache [259]. In this scheme, with even a modest number of slabs per server, a high number of combinations of $r+1$ machines will be a copyset. In other words, even a small number of simultaneous node failures in the cluster will result in data loss. When the number of slabs per server is high, almost every combination of only $r+1$ failures across the cluster will cause data loss. Therefore, to reduce the probability of data loss, we need to minimize the number of copysets, while achieving sufficient load balancing.

CodingSets: Reducing Copysets for Erasure Coding To this end, we propose CodingSets, a novel load-balancing scheme, which reduces the number of copysets for distributed erasure coding. Instead of having each node participate in several coding groups like in EC-Cache, in our scheme, each server belongs to a single, *extended* coding group. At time of coding, $(k+r)$ slabs will still be considered together, but the nodes participating in the coding group are chosen from a set of $(k+r+l)$ nodes, where l is the load-balancing factor. The nodes chosen within the extended group

are the least loaded ones. While extending the coding group increases the number of copysets (instead of $\binom{k+r}{r+1}$ copysets, now each extended coding group creates $\binom{k+r+l}{r+1}$ copysets, while the number of groups is $G = \frac{N}{k+r+l}$), it still has a significantly lower probability of data loss than having each node belong to multiple coding groups. Hydra uses CodingSets as its load balancing and slab placement policy. We evaluate it in Section 3.7.2.

Tradeoff Note that while CodingSets reduces the probability of data loss, it does not reduce the expected amount of data lost over time. In other words, it reduces the number of data loss events, but each one of these events will have a proportionally higher magnitude of data loss (i.e., more slabs will be affected) [128]. Given that our goal with Hydra is high availability, we believe this is a favorable trade off. For example, providers often provide an availability SLA, that is measured by the service available time (e.g., the service is available 99.9999% of the time). CodingSets would optimize for such an SLA, by minimizing the frequency of unavailability events.

3.6 Implementation

Resilience Manager is implemented as a loadable kernel module for Linux kernel 4.11 or later. Kernel-level implementation facilitates its deployment as an underlying block device for different remote-memory systems [92, 165, 275]. We integrated Hydra with two remote-memory systems: Infiniswap, a disaggregated VMM and Remote Regions, a disaggregated VFS. All I/O operations (e.g., slab mapping, memory registration, RDMA posting/polling, erasure coding) are independent across threads and processed without synchronization. All RDMA operations use RC and one-sided RDMA verbs (RDMA WRITE/READ). Each Resilience Manager maintains one connection for each active remote machine. For erasure coding, we use x86 AVX instructions and the ISA library [48] that achieves over 4 GB/s encoding throughput per core for (8+2) configuration in our evaluation platform.

Resource Monitor is implemented as a user-space program. It uses RDMA SEND/RECV for all control messages.

3.7 Evaluation

We evaluate Hydra on a 50-machine 56 Gbps InfiniBand CloudLab cluster against Infiniswap [165], Leap [224] (disaggregated VMM) and Remote Regions [92] (disaggregated VFS). Our evaluation addresses the following questions:

- Does it improve the resilience of cluster memory? (§3.7.1)

- Does it improve the availability? (§3.7.2)
- What is its overhead and sensitivity to parameters? (§3.7.3)
- How much TCO savings can we expect? (§3.7.4)
- What is its benefit over a persistent memory setup? (§3.7.5)

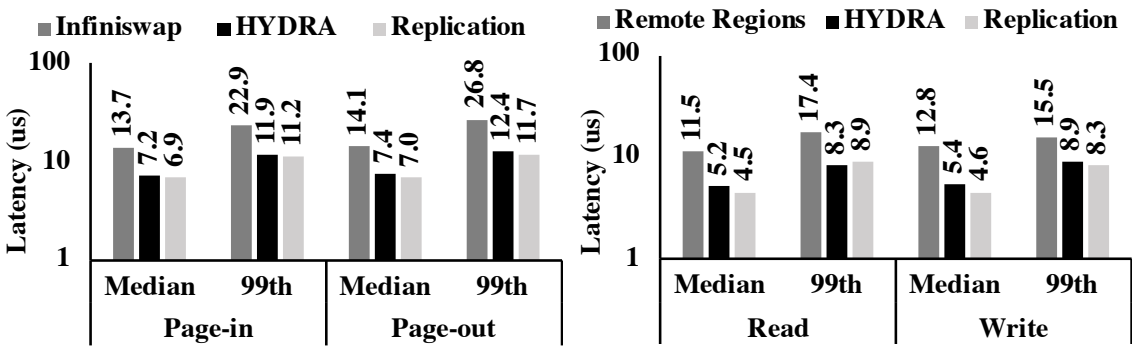
Methodology Unless otherwise specified, we use k=8, r=2, and Δ=1, targeting $1.25\times$ memory and bandwidth overhead. We select r=2 because late binding is still possible even when one of the remote slab fails. The additional read Δ=1 incurs $1.125\times$ bandwidth overhead during reads. We use 1GB $SlabSize$, The additional number of choices for eviction $E' = 2$. Free memory headroom is set to 25%, and the control period is set to 1 second. Each machine has 64 GB of DRAM and $2\times$ Intel Xeon E5-2650v2 with 32 virtual cores.

We compare Hydra against the following alternatives:

- **SSD Backup**: Each page is backed up in a local SSD for the minimum $1\times$ remote memory overhead. We consider both disaggregated VMM and VFS systems.
- **Replication**: We directly write each page over RDMA to two remote machines' memory for a $2\times$ overhead.
- **EC-Cache w/ RDMA**: Implementation of the erasure coding scheme in EC-Cache [259], but implemented on RDMA.

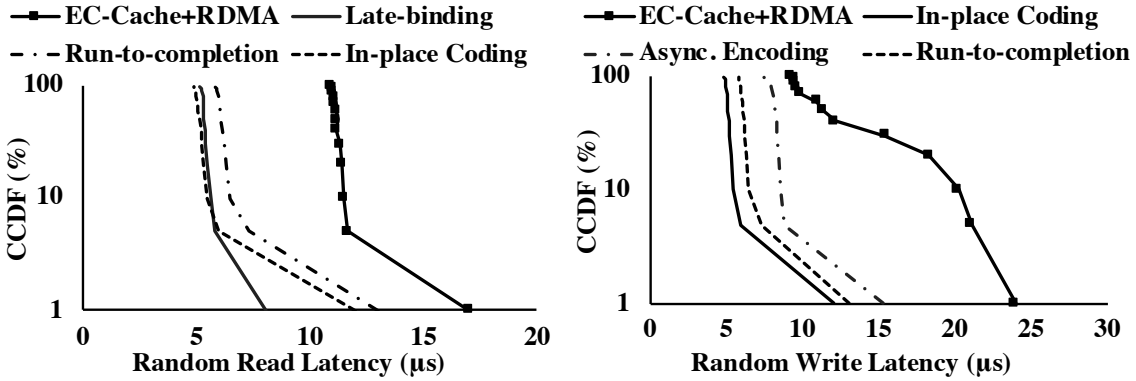
Workload Characterization Our evaluation consists of both micro-benchmarks and cluster-scale evaluations with real-world applications and workload combinations.

- We use TPC-C [79] on VoltDB [84]. We perform 5 different types of transactions to simulate an order-entry environment. We set 256 warehouses and 8 sites and run 2 million transactions. Here, the peak memory usage is 11.5 GB.
- We use Facebook's ETC, SYS workloads [104] on Memcached [56]. First, we use 10 million SETs to populate the Memcached server. Then we perform another 10 million operations (for ETC: 5% SETs, 95% GETs, for SYS: 25% SETs, 75% GETs). The key size is 16 bytes and 90% of the values are evenly distributed between 16–512 bytes. Peak memory usages are 9 GB for ETC and 15 GB for SYS.
- We use PageRank on PowerGraph [161] and Apache Spark/GraphX [162] to measure the influence of Twitter users on followers on a graph with 11 million vertices [200]. Peak memory usages are 9.5 GB and 14 GB, respectively.



(a) Disaggregated VMM Latency (b) Disaggregated VFS Latency

Figure 3.9: Hydra provides better latency characteristics during both disaggregated VMM and VFS operations.



(a) Remote Read (b) Remote Write

Figure 3.10: Hydra latency breakdown through CCDF.

3.7.1 Resilience Evaluation

We evaluate Hydra both in the presence and absence of failures with microbenchmarks and real-world applications.

3.7.1.1 Latency Characteristics

First, we measure Hydra's latency characteristics with micro-benchmarks in the absence of failures. Then we analyze the impact of its design components.

Disaggregated VMM Latency We use a simple application with its working set size set to 2GB. It is provided 1GB memory to ensure that 50% of its memory accesses cause paging. While using disaggregated memory for remote page-in, Hydra improves page-in latency over Infiniswap with

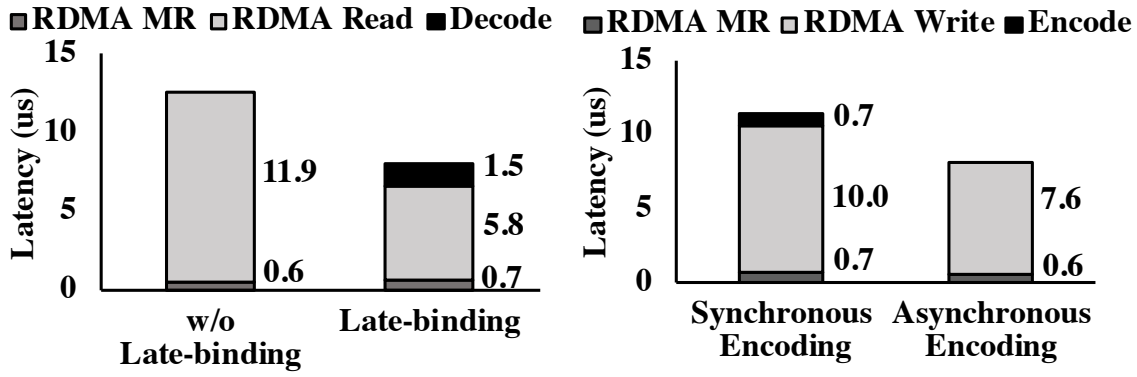
SSD backup by 1.79× at median and 1.93× at the 99th percentile. Page-out latency is improved by 1.9× and 2.2× over Infiniswap at median and 99th percentile, respectively. Replication provides at most 1.1× improved latency over Hydra, while incurring 2× memory and bandwidth overhead (Figure 3.9a).

Disaggregated VFS Latency We use `fio` [36] to generate one million random read/write requests of 4 KB block I/O. During reads, Hydra provides improved latency over Remote Regions by 2.13× at median and 2.04× at the 99th percentile. During writes, Hydra also improves the latency over Remote Regions by 2.22× at median and 1.74× at the 99th percentile. Replication has a minor latency gain over Hydra, improving latency by at most 1.18× (Figure 3.9b).

Benefit of Data Path Components Erasure coding over RDMA (i.e., EC-Cache with RDMA) performs worse than disk backup due to its coding overhead. Figure 3.10 shows the benefit of Hydra's data path components to reduce the latency.

1. Run-to-completion avoids interruptions during remote I/O, reducing the median read and write latency by 51%.
2. In-place coding saves additional time for data copying, which substantially adds up in remote-memory systems, reducing 28% of the read and write latency.
3. Late binding specifically improves the tail latency during remote read by 61% by avoiding stragglers. The additional read request increases the median latency only by 6%.
4. Asynchronous encoding hides erasure coding overhead during writes, reducing the median write latency by 38%.

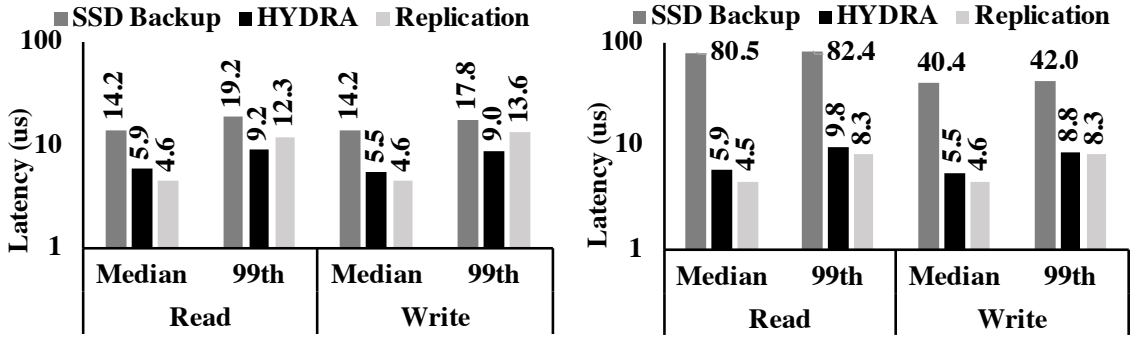
Tail Latency Breakdown The latency of Hydra consists of the time for (i) RDMA Memory Registration (MR), (ii) actual RDMA read/write, and (iii) erasure coding. Even though decoding a page takes about $1.5\mu s$, late binding effectively improves the tail latency by 1.55× (Figure 3.11a). During writes, asynchronous encoding hides encoding latency and latency impacts of straggling splits, improving tail latency by 1.34× w.r.t. synchronous encoding (Figure 3.11b). At the presence of corruption ($r = 3$), accessing extra splits increases the tail latency by 1.51× and 1.09× for reads and writes, respectively.



(a) Read breakdown

(b) Write breakdown

Figure 3.11: Hydra latency breakdown at the 99^{th} percentile.



(a) Background network flow

(b) Remote Failures

Figure 3.12: Latency in the presence of uncertainty events.

3.7.1.2 Latency Under Failures

Background Flows We generate RDMA flows on the remote machine constantly sending 1 GB messages. Unlike SSD backup and replication, Hydra ensures consistent latency due to late binding (Figure 3.12a). Hydra's latency improvement over SSD backup is 1.97–2.56×. It even outperforms replication at the tail read (write) latency by 1.33× (1.50×).

Remote Failures Both read and write latency are disk-bound when it's necessary to access the backup SSD (Figure 3.12b). Hydra reduces latency over SSD backup by 8.37–13.6× and 4.79–7.30× during remote read and write, respectively. Furthermore, it matches the performance of replication.

		TPS/OPS (thousands)		Latency (ms)			
				50th		99th	
		HYD	REP	HYD	REP	HYD	REP
VoltDB	100%	39.4	39.4	52.8	52.8	134.0	134.0
	75%	36.1	35.3	56.3	56.1	142.0	143.0
	50%	32.3	34.0	57.8	59.0	161.0	168.0
ETC	100%	123.0	123.0	123.0	123.0	257.0	257.0
	75%	119.0	125.0	120.0	121.0	255.0	257.0
	50%	119.0	119.0	118.0	122.0	254.0	264.0
SYS	100%	108.0	108.0	125.0	125.0	267.0	267.0
	75%	100.0	104.0	120.0	125.0	262.0	305.0
	50%	101.0	102.0	117.0	123.0	257.5	430.0

Table 3.2: Hydra (HYD) provides similar performance to replication (REP) for VoltDB and Memcached workloads (ETC and SYS). Higher is better for throughput; Lower is better for latency.

	Apache Spark/GraphX Completion Time (s)			PowerGraph Completion Time (s)		
	100%	75%	50%	100%	75%	50%
Hydra	77.91	105.41	191.93	73.10	66.90	68.00
Replication	77.91	91.89	195.54	73.10	73.30	73.70

Table 3.3: Hydra also provides similar completion time to replication for graph analytic applications.

3.7.1.3 Application-Level Performance

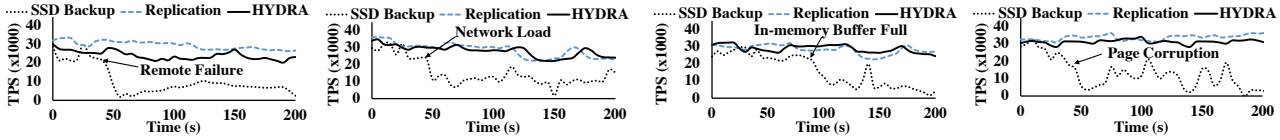
We now focus on Hydra's benefits for real-world memory-intensive applications and compare it with that of SSD backup and replication. We consider container-based application deployment [290] and run each application in an `lxc` container with a memory limit to fit 100%, 75%, 50% of the peak memory usage for each application. For 100%, applications run completely in memory. For 75% and 50%, applications hit their memory limits and performs remote I/O via Hydra.

We present Hydra's application-level performance against replication (Table 3.2 and Table 3.3) to show that it can achieve similar performance with a lower memory overhead even in the absence of any failures. For brevity, we omit the results for SSD backup, which performs much worse than both Hydra and replication – albeit with no memory overhead.

For VoltDB, when half of its data is in remote memory, Hydra achieves 0.82× throughput and almost transparent latency characteristics compared to the fully in-memory case.

For Memcached, at 50% case, Hydra achieves 0.97× throughput with read-dominant ETC workloads and 0.93× throughput with write-intensive SYS workloads compared to the 100% scenario. Here, latency overhead is almost zero.

For graph analytics, Hydra could achieve almost transparent application performance for PowerGraph; thanks to its optimized heap management. However, it suffers from increased job com-



(a) Remote failure (b) Remote Network Load (c) Request Burst (d) Page Corruption

Figure 3.13: Hydra throughput with the same setup in Figure 3.3.

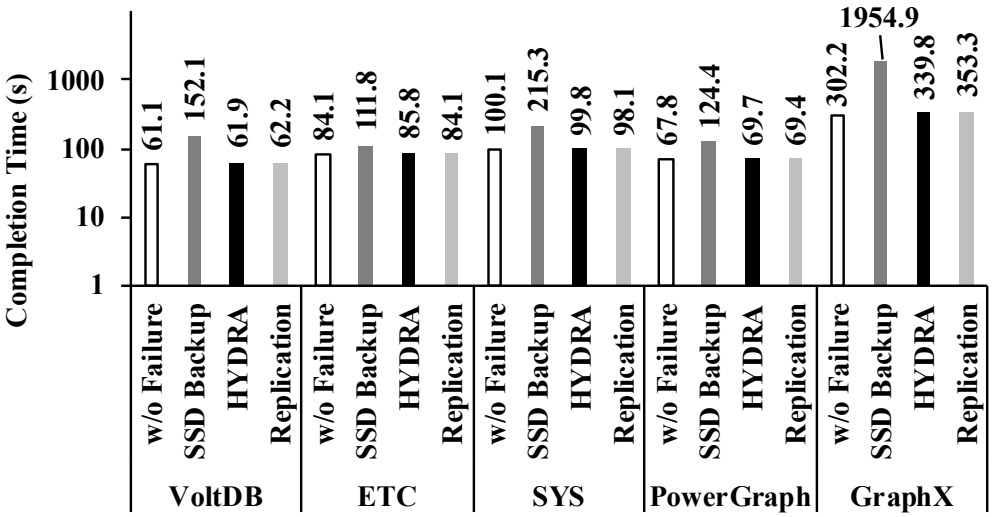


Figure 3.14: Hydra provides transparent completions in the presence of failure. Note that the Y-axis is in log scale.

pletion time for GraphX due to massive thrashing of in-memory and remote memory data – the 14 GB working set oscillates between paging-in and paging-out. This causes bursts of RDMA reads and writes. Even then, Hydra outperforms Infiniswap with SSD backup by $8.1\times$. Replication does not have significant gains over Hydra.

Performance with Leap Hydra's drop-in resilience mechanism is orthogonal to the functionalities of remote-memory frameworks. To observe Hydra's benefit even with faster in-kernel lightweight remote-memory data path, we integrate it to Leap [224] and run VoltDB and PowerGraph with 50% remote-memory configurations.

Leap waits for an interrupt during a 4KB remote I/O, whereas Hydra splits a 4KB page into smaller chunks and performs asynchronous remote I/O. Note that RDMA read for 4KB-vs-512B is 4μs-vs-1.5μs. With self-coding and run-to-completion, Hydra provides competitive performance guarantees as Leap for both VoltDB ($0.99\times$ throughput) and PowerGraph ($1.02\times$ completion time) in the absence of failures.

3.7.1.4 Application Performance Under Failures

Now we analyze Hydra's performance in the presence of failures and compare against the alternatives. In terms of impact on applications, we first go back to the scenarios discussed in Section 3.2.2 regarding to VoltDB running with 50% memory constraint. Except for the corruption scenario where we set r=3, we use Hydra's default parameters. At a high level, we observe that Hydra performs similar to replication with 1.6× lower memory overhead (Figure 3.13).

Next, we start each benchmark application in 50% settings and introduce one remote failure while it is running. We select a Resource Monitor with highest slab activities and kill it. We measure the application's performance while the Resilience Manager initiates the regeneration of affected slabs.

Hydra's application-level performance is transparent to the presence of remote failure. Figure 3.14 shows Hydra provides almost similar completion times to that of replication at a lower memory overhead in the presence of remote failure. In comparison to SSD backup, workloads experience 1.3–5.75× lower completion times using Hydra. Hydra provides similar performance at the presence of memory corruption. Completion time gets improved by 1.2–4.9× w.r.t. SSD backup.

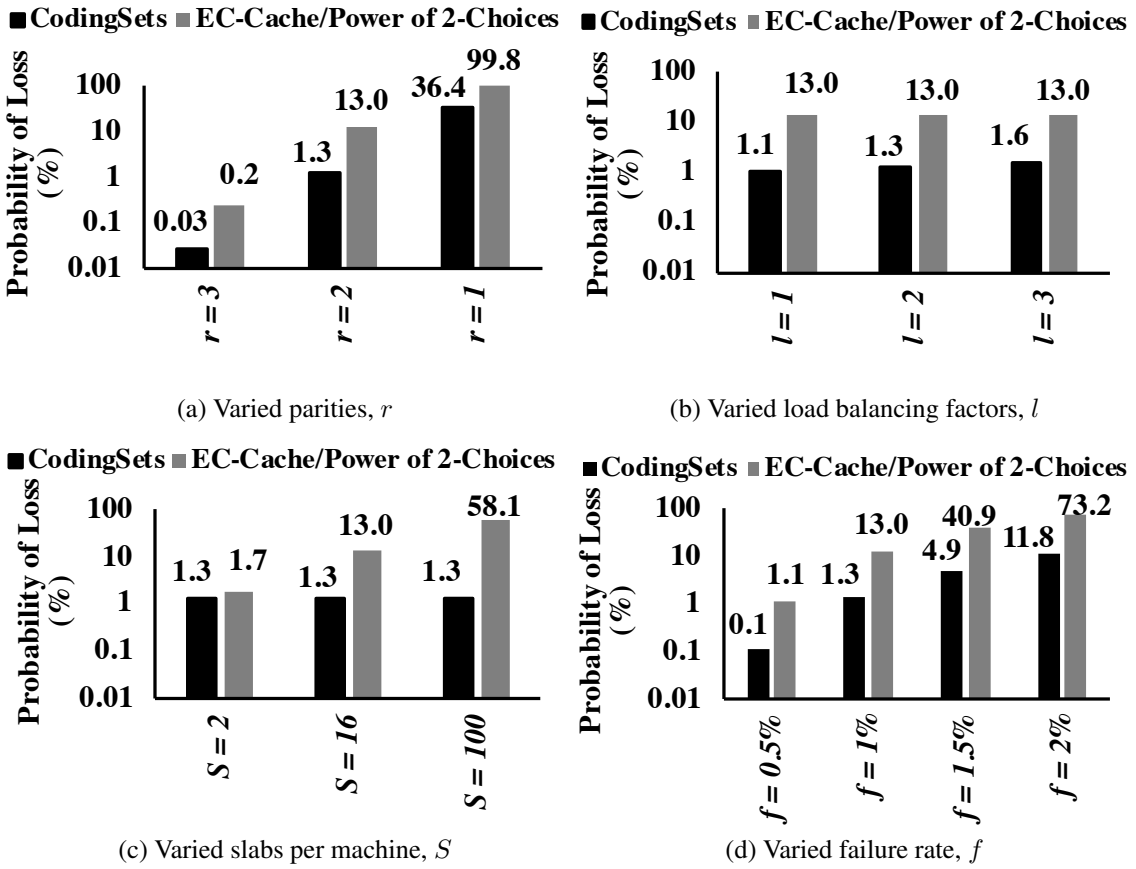
3.7.2 Availability Evaluation

In this section, we evaluate Hydra's availability and load balancing characteristics in large clusters.

3.7.2.1 Analysis of CodingSets

We compare the availability and load balancing of Hydra with EC-Cache and power-of-two-choices [233]. In CodingSets, each server is attached to a disjoint coding group. During encoded write, the $(k+r)$ least loaded nodes are chosen from a subset of the $(k+r+l)$ coding group at the time of replication. EC-Cache simply assigns slabs to coding groups comprising of random nodes. Power-of-two-choices finds two candidate nodes at random for each slab, and picks the less loaded one.

Probability of Data Loss Under Simultaneous Failures To evaluate the probability of data loss of Hydra under different scenarios in a large cluster setting, we compute the probability of data loss under the three schemes. Note that, in terms of data loss probability, we assume EC-Cache and power of two choices select random servers, and are therefore equivalent. Figure 3.15 compares the probabilities of loss for different parameters on a 1000-machine cluster. Our baseline comparison is against the best case scenario for EC-Cache and power-of-two-choices, where the number of slabs per server is low (1 GB slabs, with 16 GB of memory per server).



(a) Varied parities, r

(b) Varied load balancing factors, l

(c) Varied slabs per machine, S

(d) Varied failure rate, f

Figure 3.15: Probability of data loss at different scenarios (base parameters k=8, r=2, l=2, S=16, f=1%) on a 1000-machine cluster.

Even for a small number of slabs per server, Hydra reduces the probability of data loss by *an order of magnitude*. With a large number of slabs per server (e.g., 100) the probability of failure for EC-Cache becomes very high during correlated failure. Figure 3.15 shows that there is an inherent trade-off between the load-balancing factor (l) and the probability of data loss under correlated failures.

Load Balancing of CodingSets Figure 3.16 compares the load balancing of the three policies. EC-Cache's random selection of $(k + r)$ nodes causes a higher load imbalance, since some nodes will randomly be overloaded more than others. As a result, CodingSets improves load balancing over EC-Cache scheme by $1.1\times$ even when $l = 0$, since CodingSets' coding groups are non-overlapping. For $l = 4$, CodingSets provides with $1.5\times$ better load balancing over EC-Cache at 1M machines. The power of two choices improves load balancing by 0%-20% compared CodingSets with $l = 2$, because it has more degrees of freedom in choosing nodes, but suffers from an order of magnitude higher failure rate (Figure 3.15).

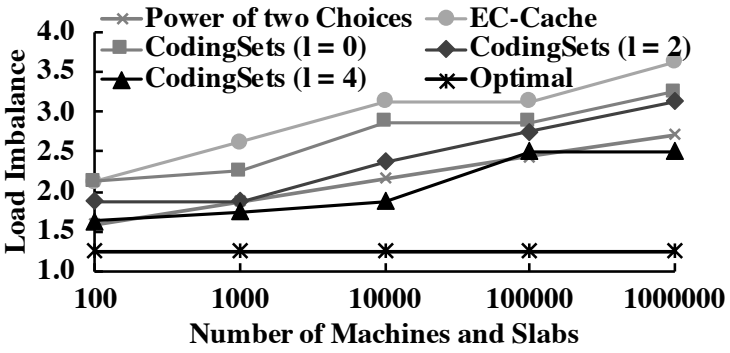
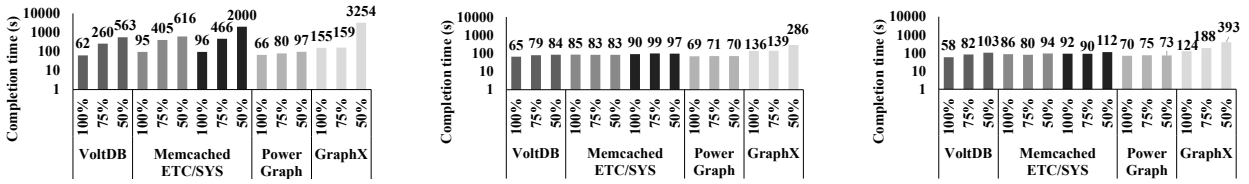


Figure 3.16: CodingSets enhances Hydra with better load balancing across the cluster (base parameters k=8, r=2).



(a) SSD Backup (b) Hydra (c) Replication

Figure 3.17: Median completion times (i.e., throughput) of 250 containers on a 50-machine cluster.

3.7.2.2 Cluster Deployment

We run 250 containerized applications across 50 machines. For each application and workload, we create a container and randomly distribute it across the cluster. Here, total memory footprint is 2.76 TB; our cluster has 3.20 TB of total memory. Half of the containers use 100% configuration; about 30% use the 75% configuration; and the rest use the 50% configuration. There are at most two simultaneous failures.

Application Performance We compare application performance in terms of completion time (Figure 3.17) and latency (Table 3.4) that demonstrate Hydra's performance benefits in the presence of cluster dynamics. Hydra's improvements increase with decreasing local memory ratio. Its throughput improvements w.r.t. SSD backup were up to 4.87× for 75% and up to 20.61× for 50%. Its latency improvements were up to 64.78× for 75% and up to 51.47× for 50%. Hydra's performance benefits are similar to replication (Figure 3.17c), but with lower memory overhead.

Impact on Memory Imbalance and Stranding Figure 3.18 shows that Hydra reduces memory usage imbalance w.r.t. coarser-grained memory management systems: in comparison to SSD backup-based (replication-based) systems, memory usage variation decreased from 18.5% (12.9%)

Latency (ms)		50th			99th		
		SSD	HYD	REP	SSD	HYD	REP
VoltDB	100%	55	60	48	179	173	177
	75%	60	57	48	217	185	225
	50%	78	61	48	305	243	225
ETC	100%	138	119	118	260	245	247
	75%	148	113	120	9912	240	263
	50%	167	117	111	10175	244	259
SYS	100%	145	127	125	249	269	267
	75%	154	119	113	17557	271	321
	50%	124	111	117	22828	452	356

Table 3.4: VoltDB and Memcached (ETC, SYS) latencies for SSD backup, Hydra (HYD) and replication (REP) in cluster setup.

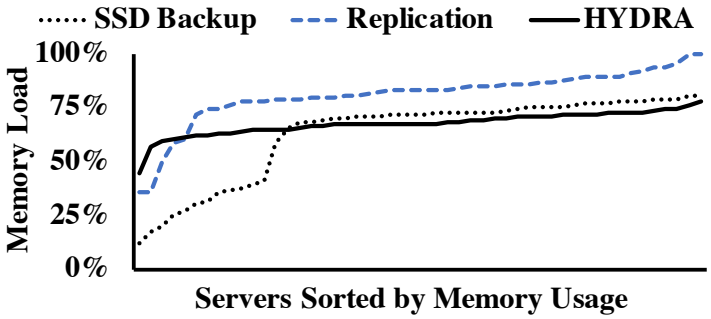


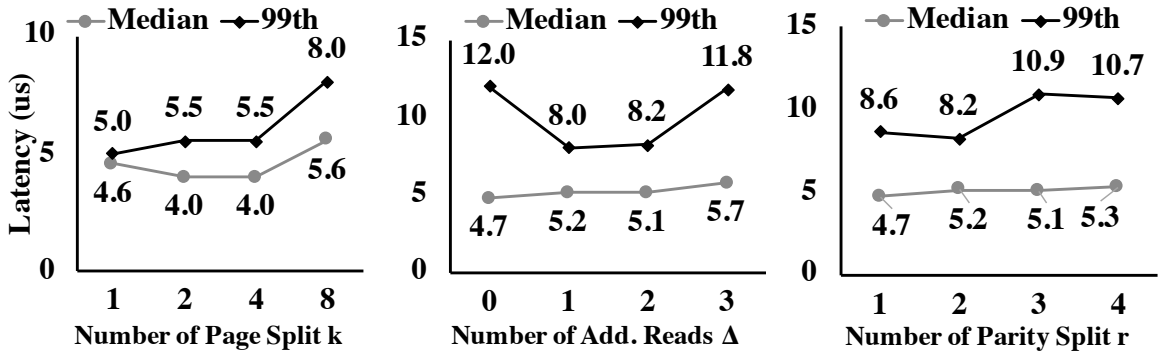
Figure 3.18: Average memory usage across 50 servers.

to 5.9% and the maximum-to-minimum utilization ratio decreased from 6.92× (2.77×) to 1.74×. Hydra better exploits unused memory in under-utilized machines, increasing the minimum memory utilization of any individual machine by 46%. Hydra incurs about 5% additional total memory usage compared to disk backup, whereas replication incurs 20% overhead.

3.7.3 Sensitivity Evaluation

Impact of (k, r, Δ) Choices Figure 3.19a shows read latency characteristics for varying k. Increasing from k=1 to k=2 reduces median latency by parallelizing data transfers. Further increasing k improves space efficiency (measured as $\frac{r}{k+r}$) and load balancing, but latency deteriorates as well.

Figure 3.19b shows read latency for varying values of Δ. Although just one additional read (from Δ=0 to Δ=1) helps tail latency, more additional reads have diminishing returns; instead, it hurts latency due to proportionally increasing communication overheads. Figure 3.19c shows write latency variations for different r values. Increasing r does not affect the median write latency. However, the tail latency increases from $r = 3$ due to the increase in overall communication overheads.



(a) r=4, Δ=1 (b) k=8, r=4 (c) k=8, Δ=1

Figure 3.19: Impact of page splits (k), additional reads (Δ) on read latency, and parity splits (r) on write latency.

Monthly Pricing	Google	Amazon	Microsoft
Standard machine	$1,553	$2,304	$1,572
1% memory	$5.18	$9.21	$5.92
Hydra	6.3%	8.4%	7.3%
Replication	3.3%	4.8%	3.9%
PM Backup	3.5%	7.6%	4.9%

Table 3.5: Revenue model and TCO savings over three years for each machine with 30% unused memory on average.

Resource Overhead We measure average CPU utilization of Hydra components during remote I/O. Resilience Manager uses event-driven I/O and consumes only 0.001% CPU cycles in each core. Erasure coding causes 0.09% extra CPU usage per core. As Hydra uses one-sided RDMA, remote Resource Monitors do not have CPU overhead in the data path.

In cluster deployment, Hydra increases CPU utilization by 2.2% on average and generates 291 Mbps RDMA traffic per machine, which is only 0.5% of its 56 Gbps bandwidth. Replication has negligible CPU usage but generates more than 1 Gbps traffic per machine.

Background Slab Regeneration To observe the overall latency to regenerate a slab, we manually evict one of the remote slabs. When it is evicted, Resilience Manager places a new slab and provides the evicted slab information to the corresponding Resource Monitor, which takes 54 ms. Then the Resource Monitor randomly selects k out of remaining remote slabs and read the page data, which takes 170 ms for a 1 GB slab. Finally, it decodes the page data to the local memory slab within 50 ms. Therefore, the total regeneration time for a 1 GB size slab is 274 ms, as opposed to taking several minutes to restart a server after failure.

To observe the impact of slab regeneration on disaggregated VMM, we run the micro-

System	Year	Deployability	Fault Tolerance	Load Balancing	Latency Tolerance
Memory Blade [215]	'09	HW Change	Reprovision	None	None
RamCloud [248]	'10	App. Change	Remote Disks	Power of Choices	None
FaRM [142]	'14	App. Change	Replication	Central Coordinator	None
EC-Cache [259]	'16	App. Change	Erasure Coding	Multiple Coding Groups	Late Binding
Infiniswap [165]	'17	Unmodified	Local Disk	Power of Choices	None
Remote Regions [92]	'18	App. Change	None	Central Manager	None
LegoOS [275]	'18	OS Change	Remote Disk	None	None
Compressed Far Memory [203]	'19	OS Change	None	None	None
Leap [224]	'20	OS Change	None	None	None
Kona [111]	'21	HW Change	Replication	None	None
Hydra		Unmodified	Erasure Coding	CodingSets	Late Binding

Table 3.6: Selected proposals on remote memory in recent years.

benchmark mentioned in §3.7.1. At the half-way of the application's runtime, we evict one of the remote slabs. Background slab regeneration has a minimal impact on the remote read – remote read latency increases by 1.09×. However, as remote writes to the victim slab halts until it gets regenerated, write latency increases by 1.31×.

3.7.4 TCO Savings

We limit our TCO analysis only to memory provisioning. The TCO savings of Hydra is the revenue from leveraged unused memory after deducting the TCO of RDMA hardware. We consider capital expenditure (CAPEX) of acquiring RDMA hardware and operational expenditure (OPEX) including their power usage over 3 years. An RDMA adapter costs $600 [54], RDMA switch costs $318 [55] per machine, and the operating cost is $52 over 3 years [165] – overall, the 3-year TCO is $970 for each machine. We consider the standard machine configuration and pricing from Google Cloud Compute [42], Amazon EC2 [7], and Microsoft Azure [7] to build revenue models and calculate the TCO savings for 30% of leveraged memory for each machine (Table 3.5). For example, in Google, the savings of disaggregation over 3 years using Hydra is (($5.18*30*36)/1.25-$970)/($1553*36)*100% = 6.3%.

3.7.5 Disaggregation with Persistent Memory Backup

To observe the impact of persistent memory (PM), we run all the micro-benchmarks and real-world applications mentioned earlier over Infiniswap with local PM backup. Unfortunately, at the time of writing, we cannot get hold of a real Intel Optane DC. We emulate PM using DRAM with the latency characteristics mentioned in prior work [133].

Replacing SSD with local PM can significantly improve Infiniswap's performance in a disaggregated cluster. However, for the micro-benchmark mentioned in §3.7.1, Hydra still provides 1.06× and 1.09× better 99th percentile latency over Infiniswap with PM backup during page-in and page-out, respectively. Even for real-world applications mentioned in §3.7.1.3, Hydra almost matches the performance of local PM backup – application-level performance varies within 0.94–1.09× of that with PM backup. Note that replacing SSD with PM throughout the cluster does not improve the availability guarantee in the presence of cluster-wide uncertainties. Moreover, while resiliency through unused remote DRAM is free, PM backup costs $11.13/GB [65]. In case of Google, the additional cost of $2671.2 per machine for PM reduces the savings of disaggregation over 3 years from 6.3% to (($5.18*30*36)-$970-$2671.2)/($1553*36)*100% = 3.5% (Table 3.5).

3.8 Related Work

Remote-Memory Systems Many software systems tried leveraging remote machines' memory for paging [151, 223, 241, 120, 145, 211, 3, 270, 165, 203, 224, 111], global virtual memory abstraction [150, 75, 198], and to create distributed data stores [248, 142, 212, 197, 289, 26, 180, 267]. Hardware-based remote access to memory using PCIe interconnects [215] and extended NUMA fabric [245] are also proposed. Table 3.6 compares a selected few.

Cluster Memory Solutions With the advent of RDMA, there has been a renewed interest in cluster memory solutions. The primary way of leveraging cluster memory is through key-value interfaces [231, 181, 142, 248], distributed shared memory [254, 239], or distributed lock [311]. However, these solutions are either limited by their interface or replication overheads. Hydra, on the contrary, is a transparent, memory-efficient, and load-balanced mechanism for resilient remote memory.

Erasure Coding in Storage Erasure coding has been widely employed in RAID systems to achieve space-efficient fault tolerance [325, 271]. Recent large-scale clusters leverage erasure coding for storing *cold* data in a space-efficient manner to achieve fault-tolerance [300, 171, 237]. EC-Cache [259] is an erasure-coded in-memory cache for 1MB or larger objects, but it is highly susceptible to data loss under correlated failures, and its scalability is limited due to communication overhead. In contrast, Hydra achieves resilient erasure-coded remote memory with single-digit μs page access latency.

3.9 Conclusion

Hydra leverages online erasure coding to achieve single-digit μs latency under failures, while judiciously placing erasure-coded data using CodingSets to improve availability and load balancing. It matches the resilience of replication with 1.6$\times$ lower memory overhead and significantly improves latency and throughput of real-world memory-intensive applications over SSD backup-based resilience. Furthermore, CodingSets allows Hydra to reduce the probability of data loss under simultaneous failures by about 10$\times$. Overall, Hydra makes resilient remote memory practical.

CHAPTER 4

Memtrade: Marketplace for Disaggregated Memory Clouds

4.1 Introduction

Cloud resources are increasingly being offered in an elastic and disaggregated manner. Examples include serverless computing [15, 20] and disaggregated storage [9, 190, 292, 191, 158] that scale rapidly and adapt to highly dynamic workloads [236, 193, 204, 192, 5]. Memory, however, is still largely provisioned statically, especially in public cloud environments. In public clouds, a user launching a new VM typically selects from a set of static, pre-configured instance types, each with a fixed number of cores and a fixed amount of DRAM [8, 83, 53]. Although some platforms allow users to customize the amount of virtual CPU and DRAM [29], the amount remains static throughout the lifetime of the instance. Even in serverless frameworks, which offer elasticity and auto-scaling, a function has a static limit on its allocation of CPU and memory [16].

At the same time, long-running applications deployed on both public and private clouds are commonly highly over-provisioned relative to their typical memory usage. For example, cluster-wide memory utilization in Google, Alibaba, and Meta datacenters hovers around 40%–60% [263, 275, 165, 268]. Large-scale analytics service providers that run on public clouds, such as Snowflake, fare even worse – on average 70%–80% of their memory remains unutilized [292]. Moreover, in many real-world deployments, workloads rarely use all of their allocated memory all of the time. Often, an application allocates a large amount of memory but accesses it infrequently (§4.2.2). For example, in Google's datacenters, up to 61% of allocated memory remains idle [202]. In Meta's private datacenters, within a 10-minutes window, applications use only 30–60% of the allocated memory [225]. Since DRAM is a significant driver of infrastructure cost and power consumption [146, 34, 147, 225], excessive underutilization leads to high capital and operating expenditures, as well as wasted energy (and carbon emissions). Although recent remote memory systems address this by satisfying an application's excess memory demand from an underutilized server [165, 202, 92, 224, 267, 99, 111], existing frameworks are designed for private datacenters.

Furthermore, with the emergence of coherent interfaces like Compute Express Link (CXL) [27], next-generation datacenter designs are moving towards tiered-memory subsystems [225, 208]. Servers within a rack can be connected through CXL switches and access each other memory. In such a system, CPUs of one server will have access to heterogeneous memory types with varied latency, bandwidth, and performance characteristics (Figure 1.3). While running multiple applications in such a rack-scale system, system-wide application-level performance will highly depend on an application's share to different memory tiers. Efficiently rightsizing different memory tiers, moving cold pages from faster to slower memory tiers, and matching harvested memories on different tiers to appropriate applications with a view to ensuring performance is challenging in such a disaggregated system.

In this paper, we harvest both unallocated and allocated-but-idle application memory to enable remote memory in public clouds. We propose a new memory consumption model that allows overprovisioned and/or idle applications (*producers*) to offer excess idle memory to memory-intensive applications (*consumers*) that are willing to pay for additional memory for a limited period of time at an attractive price, via a trusted third-party (*broker*). Participation is voluntary, and either party can leave at any time. Practical realization of this vision must address following challenges:

- *Immediately Deployable.* Our goal is that Memtrade is immediately deployable on existing public or private clouds. Prior frameworks depend on host kernel or hypervisor modifications [165, 275, 202, 92, 224, 99]. In many cases (e.g., public cloud setting, server configuration restrictions), a tenant cannot modify host-level system software which would require the operator to manage the remote memory service. In addition, prior work assumes the latest networking hardware and protocols (e.g., RDMA) [165, 275, 202, 92, 224, 267, 111]; availability of these features in public clouds is limited, restricting adoption. Memtrade being an ubiquitous solution, allows users hop on any existing machine and readily deploy it without disrupting the running application(s) and modifying the underline kernel.

- *Efficient Harvesting.* Memory harvesting needs to be lightweight, transparent and easily deployable without impacting performance. Most prior work includes only a VM's *unallocated* memory in the remote memory pool. Leveraging idle application-level memory – allocated to an application but later unused or accessed infrequently – significantly enhances remote memory capacity. This is especially challenging in public clouds, where a third-party provider has limited visibility of tenant workloads, and workloads may shift at any time. Existing *cold page detection*-based [202, 88] proactive page reclamation techniques need significant CPU and memory resources, along with host kernel or hypervisor modifications [225].

- *Performant Consumption.* To ensure producer-side performance, Memtrade must return

a producer's harvested memory seamlessly when needed. Memory offered to consumers may also disappear due to a sudden burst in the producer's own memory demand, or if a producer leaves unexpectedly. Memtrade needs to manage this unavailability to provide a high-performance memory interface. Besides, modern datacenters are going to observe an architectural paradigm shift towards memory pooling and disaggregation where efficiently migrating hot/cold memories to appropriate memory tiers will be a major concern [225, 208]. Memtrade harvester needs to be aware of the impact of different memory tiers. Effectively harvesting free spaces on the faster memory tiers while moving colder pages to the slower tiers can help tiered-memory subsystems.

- *Incentivization and Resource Matching.* Unlike prior work, which assumes cooperative applications, in a public cloud setting, we need to create a *market* where producers and consumers have monetary incentives to participate. Producers must be compensated for leasing memory, and the price must be attractive to consumers compared to alternatives (e.g., existing in-memory caching services or spot instances). In addition, producers have varied availability and bursty workload demands, while consumers may have their own preferences regarding remote memory availability, fragmentation, network overhead, and application-level performance, all which must be considered when matching producers to consumers.

We design and develop *Memtrade*, an immediately-deployable realization of remote memory on public clouds that addresses these challenges without any host kernel or hypervisor modifications. Memtrade employs a *harvester* in each producer VM to monitor its application-level performance and adaptively control resource consumption.The harvester uses an adaptive control loop that decides when to harvest from and when to return memory to the producer.

To prevent performance degradation in memory-sensitive applications, we design a novel in-memory swap space, *Silo*, which serves as a temporary victim cache for harvested pages. In the case of a sudden loss of performance, the harvester proactively prefetches previously-harvested pages back into memory. The combination of these mechanisms allows Memtrade to harvest idle pages with low impact to producer workload performance and offer them to consumers. Consumers of Memtrade can access the harvested memory through a key-value (KV) cache or a swap interface. For public cloud adaptation or secure memory consumption, Memtrade consumer interfaces provide optional plugable cryptographic protections for the confidentiality and integrity of data stored in the untrusted producer VM.

To maximize cluster-wide utilization, Memtrade employs a *broker* – a central coordinator that manages the remote-memory market and matches producers and consumers based on their supply and demand, and helps facilitate their direct communication. The broker sets the price per unit of remote memory and is incentivized by receiving a cut of monetary transactions. The broker does

not require any support from the cloud provider, and it does not need to economically benefit the cloud providers to be viable. Of course, cloud providers may choose to support it, but it's optional. One can implement their own pricing strategies based on different objectives; Memtrade broker service is orthogonal towards the pricing and matching mechanisms.

Although we focus primarily on virtualized public clouds, Memtrade can be deployed in other settings, such as private datacenters and containerized clouds. We plan to open-source Memtrade. Overall, we make the following contributions:

- Memtrade is the first end-to-end system that enables a disaggregated memory marketplace on clouds (§4.3). Memtrade is easily-deployable without any support from the cloud provider.
- We design a system to identify and harvest idle memory with minimal overhead and negligible performance impact (§4.4), which uses Silo – a novel in-memory victim cache for swapped-out pages to reduce performance loss during harvesting.
- We design a broker that arbitrates between consumers and producers, enables their direct communication and implements a placement policy based on consumer preferences, fragmentation, and producer availability (§4.5).
- Memtrade improves consumer average latency by up to 2.8×, while impacting producer latency by less than 2.1% (§5.6), and improves memory utilization (up to 97.9%).

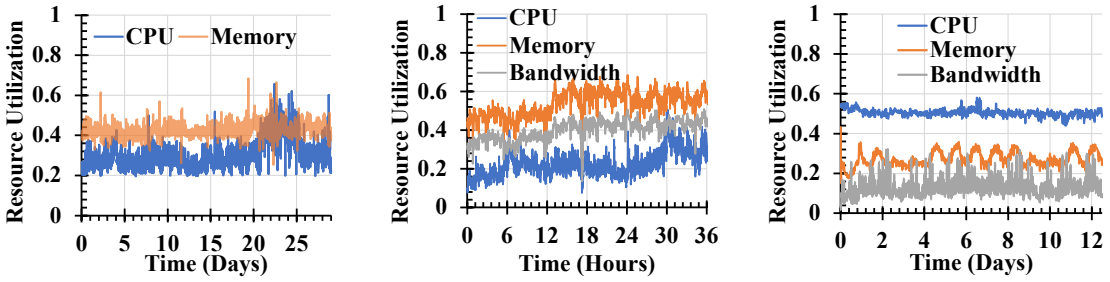
4.2 Background and Motivation

4.2.1 Remote Memory

Remote memory exposes capacity available in remote hosts as a pool of memory shared among many machines. It is often implemented logically by leveraging unallocated memory in remote machines via well-known abstractions, such as files [92], remote memory paging [165, 157, 211, 99, 111], distributed OS virtual memory management [275] and the C++ Standard Library data structures [267]. Existing frameworks require specialized kernels, hypervisors or hardware that might not be available in public clouds. Prior works focus on private-cloud use cases [202, 165, 92, 99, 267, 111] and do not consider the transient nature of public-cloud remote memory, nor the isolation and security challenges when consumers and producers belong to different organizations.

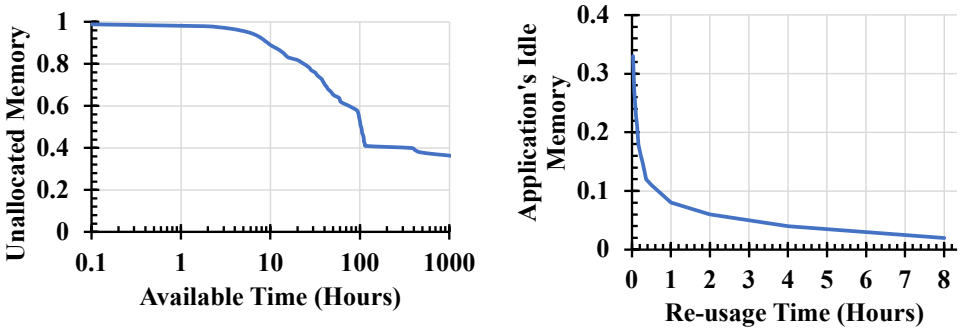
4.2.2 Resource Underutilization in Cloud Computing

Underutilized Resources. Due to over-provisioning, a significant portion of resources remains idle in private and public clouds that run a diverse mix of workloads. To demonstrate this, we



(a) **Google Cluster** (b) **Alibaba Cluster** (c) **Snowflake Cloud**

Figure 4.1: Cluster resources remain significantly unallocated in (a) Google, (b) Alibaba, and (c) Snowflake.

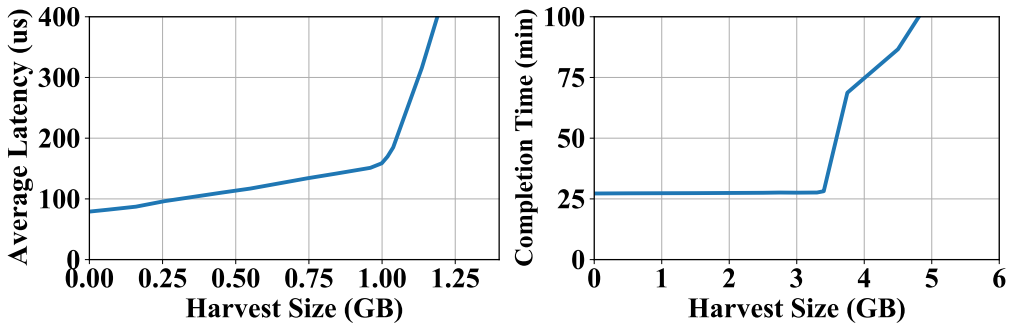


(a) **Google Cluster Memory** (b) **Idle Application Memory**

Figure 4.2: (a) Unallocated memory remains available for long periods, but (b) idle memory are reused quickly.

analyze production traces of Google [41], Alibaba [4], and Snowflake [292] clusters for periods of 29 days, 36 hours, and 14 days, respectively (Figure 4.1). In Google's cluster, averaging over one-hour windows, memory usage never exceeds 60% of cluster capacity. In Alibaba's cluster, at least 30% of the total memory capacity always remains unused. Even worse, in Snowflake's cluster, which runs on public clouds, 80% of memory is unutilized on average.

However, idle memory alone is not sufficient for providing remote memory access; in the absence of dedicated hardware such as RDMA, it also requires additional CPU and network-bandwidth both at the consumer and the producer. Fortunately, the production traces show that a significant portion of these resources are underutilized. Approximately, 50–85% of cluster CPU capacity remains idle in all of these traces; Alibaba and Snowflake traces, which include bandwidth usage, show that 50–75% of network capacity remains idle.



(a) **Zipfian on Redis** (b) **XGBoost**

Figure 4.3: Performance drop while harvesting memory for (a) Zipfian trace on Redis, and (b) XGBoost training.

Availability of Unallocated and Idle Memory. Another important consideration is whether unutilized memory remains available for sufficiently long periods of time to enable other applications to access it productively. Figure 4.2a shows that 99% of the unallocated memory in the Google cluster remains available for at least an hour. Beyond unallocated memory, which constitutes 40% of the memory in the Google traces, a significant pool of memory is allocated to applications but remains idle [99]. Figure 4.2b shows that an additional 8% of total memory is application memory that is not touched for an hour or more. In public clouds, where many tenants are over-provisioned, the proportion of application idle memory may be much higher [292].

Uses for Transient Remote Memory. Transient remote memory seems attractive for numerous uses in many environments. KV caches are widely used in cloud applications [242, 129, 86, 105, 308, 109], and many service providers offer popular in-memory cache-as-a-service systems [10, 57, 68, 19]. Similarly, transient memory can be used for filesystem-as-a-service [328] in serverless computing. Application developers routinely deploy remote caches in front of persistent storage systems, and in-memory caches are a key driver of memory consumption in clouds [242, 105].

4.2.3 Disaggregation Challenges in Public Clouds

Harvesting Application Memory. Beyond unallocated memory, a large amount of potentially-idle memory is allocated to user VMs. In many cases, harvesting such idle memory has minimal performance impacts. However, harvesting too aggressively can result in severe performance degradation, or even crash applications. Figure 4.3 shows the performance degradation while harvesting memory from two applications. We can harvest a substantial amount of memory from each without much performance loss. However, performance can quickly fall off a cliff, and dynamic application load changes necessitate adaptive harvesting decisions in real-time.

To reclaim an application's idle memory, existing solutions use kernel modifications [202, 88, 225] to determine the age of pages mapped by the application. A well-explored technique is to periodically scan the *accessed bit* present in page table entries to infer if a physical page is accessed in a given time period. If a page's age goes beyond a threshold, then it is marked as a cold page and, therefore, reclaimed. Such an approach is difficult to deploy in public clouds, where each user controls its own kernel distribution. Moreover, continuous page tracking can consume significant CPU and memory and require elevated permissions from the host kernel [45].

Transience of Remote Memory. Producers and consumers have their own supply and demand characteristics. At the same time, producers may disappear at any time, and the amount of unallocated and idle memory that can be harvested safely from an application varies over time. Given the remote I/O cost, too much churn in memory availability may deteriorate a consumer's performance.

Security. VMs that do not belong to the same organization in the public cloud are completely untrusted. Since consumer data residing in remote memory may be read or corrupted due to accidents or malicious behavior, its confidentiality and integrity must be protected. Producer applications must also be protected from malicious remote memory operations, and the impact of overly-aggressive or malicious consumers on producers must be limited.

4.3 Memtrade: Overview

Memtrade is a system that realizes remote memory on existing clouds. It consists of three core components (Figure 4.4): **(i) producers**, which expose their harvested idle memory to the remote-memory market (§4.4); **(ii) the broker**, which pairs producers with consumers while optimizing cluster-wide objectives, such as maximizing resource utilization (§4.5); and **(iii) consumers**, which request remote-memory allocations based on their demand and desired performance characteristics (§4.6). This section provides an overview of these components and their interactions; more details appear in subsequent sections.

Producers. A producer employs a collection of processes to harvest idle memory within a VM, making it available to the remote-memory market. A producer voluntarily participates in the market by first registering with the broker. The producer then monitors its resource usage and application-level performance metrics, periodically notifying the broker about its resource availability. The producer harvests memory slowly until it detects a possible performance degradation,

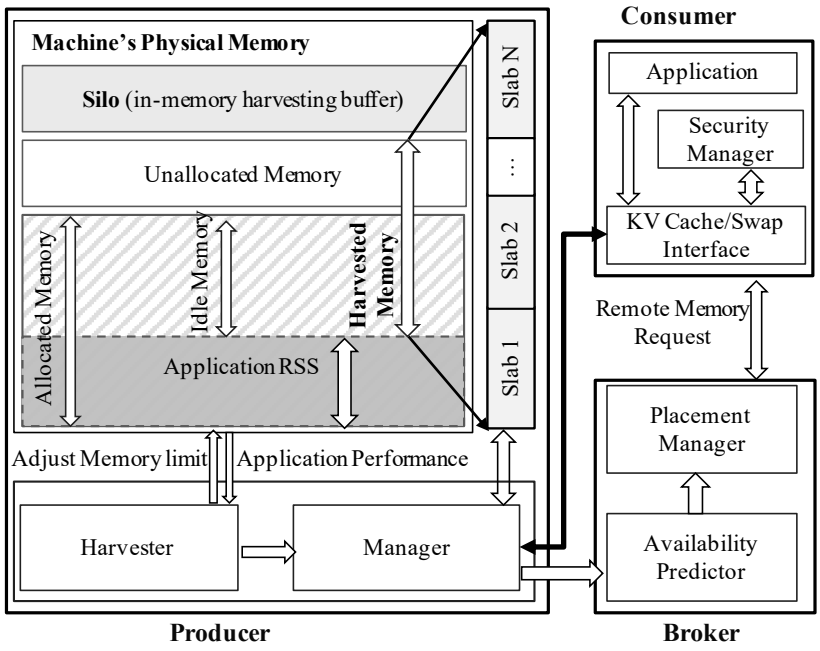


Figure 4.4: Memtrade architecture overview.

causing it to back off and enter recovery mode. During recovery, memory is returned to the producer application proactively until its original performance is restored. When it is safe to resume harvesting, the producer transitions back to harvesting mode.

When the broker matches a consumer's remote memory request to the producer, it is notified with the consumer's connection credentials and the amount of requested memory. The producer then exposes harvested memory through fixed-sized slabs dedicated to that consumer. A producer may stop participating at any time by deregistering with the broker.

Broker. The broker arbitrates between producers and consumers, matching supply and demand for harvested remote memory while considering consumer preferences and constraints. While Memtrade supports untrusted producers and consumers from diverse tenants, its logically-centralized broker component should be run by a trusted third party – such as a caching-as-a-service provider [57, 68] or the public cloud operator. The broker facilitates the direct connection between the consumer and producer using existing virtual private cloud interconnection tools (e.g.AWS Transit Gateway), which allow the consumer and producer to communicate even if they are on separate virtual private networks [18, 39]. Note that cloud providers already support this capability. The broker decides on the per-unit remote memory price for a given lease time, based on monitoring the current price of spot instances offered in the same public cloud. Appropriate pricing provides incentives for both producers and consumers to participate in the market; the broker

receives a cut of the monetary transactions it brokers as commission.

To improve the availability of transient remote memory, the broker relies on historical resource usage information for producers to predict their future availability. It additionally considers producer reputations, based on the frequency of any prior broken leases, in order to reduce occurrences of unexpected remote memory revocations. Finally, it assigns producers to consumers in a manner that maximizes the overall cluster-wide resource utilization.

Consumers. A consumer voluntarily participates in the remote-memory market by registering its connection credentials with the broker. Once approved by the broker, the consumer can submit a remote memory request by specifying its required remote memory, lease time, and preferences. After matching the request with one or more producers, the broker sends a message to the consumer with connection credentials for the assigned producer(s).

The consumer then communicates directly with assigned producers through a simple KV cache `GET` / `PUT` / `DELETE` interface to access remote memory. We also implement a transparent remote-paging interface for the consumer. When memory will be occasionally evicted by producers, and a the data needs to be stored persistently, a swap interface may face performance issues. Conveniently, applications using caches assume that data is not persistent, and may be evicted asynchronously. To ensure confidentiality and integrity of consumer data stored in producer memory, consumer interfaces offer an optional cryptographically access to remote memory in a transparent manner (§4.6.1).

4.4 Producer

The producer consists of two key components: the *harvester*, which employs a control loop to harvest application memory, and the *manager*, which exposes harvested memory to consumers as remote memory. The producer does not require modifying host-level software, facilitating deployment in existing public clouds. Our current producer implementation only supports Linux VMs. The harvester coordinates with a loadable kernel module within the VM to make harvesting decisions, without recompiling the guest kernel.

The harvester runs producer applications within a Linux control group (cgroup) [23] to monitor and limit the VM's consumption of resources, including DRAM and CPU; network bandwidth is managed by a custom traffic shaper (§4.4.2). Based on application performance, the harvester decides whether to harvest more memory or release already-harvested memory. Besides unallocated memory which is immediately available for consumers, the harvester can increase the free memory within the VM by reducing the resident set size (RSS) of the application. In *harvesting mode*, the cgroup limit is decreased incrementally to reclaim memory in relatively small chunks; the default

Algorithm 3 Harvester Pseudocode

```
1: procedure DoHarvest
2:     Decrease cgroup memory limit by ChunkSize
3:     sleep(CoolingPeriod)                          ▷ wait for performance impact
4: procedure DoRecovery
5:     while RecoveryPeriod not elapsed do
6:         Disable cgroup memory limit
7: procedure RunHarvester
8:     for each performance monitor epoch do
9:         if no page-in then
10:            Add performance data point to baseline estimator
11:            Generate baseline performance distribution
12:        Generate recent performance distribution
13:        if performance drop detected then
14:            DoRecovery()
15:        else
16:            DoHarvest()
17:        if severe performance drop detected then
18:            Prefetch from disk
```

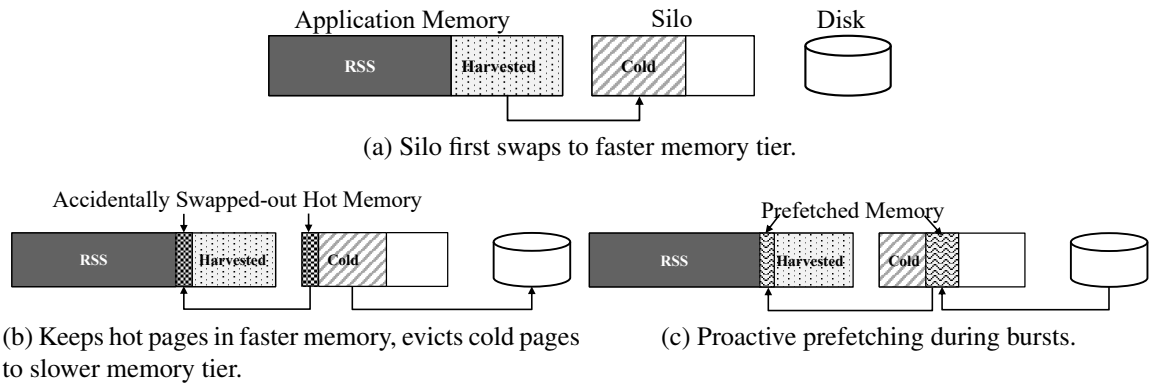
`ChunkSize` is 64 MB. If a performance drop is detected, the harvester stops harvesting and enters *recovery mode*, disabling the cgroup memory limit and allowing the application to fully recover.

Because directly reclaiming memory from an application address space can result in performance cliffs if hot pages are swapped to disk, we introduce *Silo*, a novel in-memory region that serves as a temporary buffer, or victim cache, holding harvested pages before they are made available as remote memory. Silo allows the harvester to return recently-reclaimed pages to applications efficiently. In addition, when it detects a significant performance drop due to unexpected load, Silo proactively prefetches swapped-out pages from disk, which helps mitigate performance cliffs. Algorithm 3 presents a high-level sketch of the harvester's behavior.

The manager exposes the harvested memory via a key-value cache `GET` / `PUT` / `DELETE` interface, by simply running a Redis server for each consumer. A key challenge for the manager is handling the scenario where the harvester needs to evict memory. We leverage the existing Redis LRU cache-eviction policy, which helps reduce the impact on consumers when the producer suddenly needs more memory.

4.4.1 Adaptive Harvesting of Remote Memory

Monitoring Application Performance. The harvester provides an interface for applications to periodically report their performance, with a metric such as latency or throughput. Without loss of generality, our description uses a performance metric where higher values are better. Many



(a) Silo first swaps to faster memory tier.

(b) Keeps hot pages in faster memory, evicts cold pages to slower memory tier.

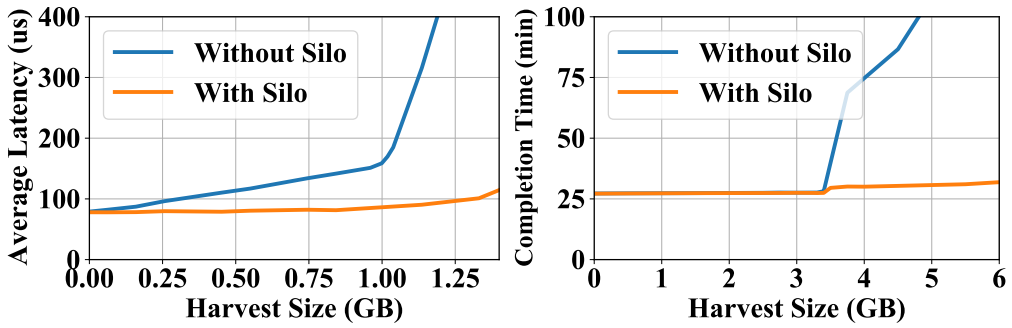
(c) Proactive prefetching during bursts.

Figure 4.5: Memory harvesting and recovery using Silo.

applications already expose performance metrics. For example, applications often periodically write their throughput, latency, network bandwidth usage, etc. metrics to a log structure. From this information, we can directly understand the performance variations overtime. For applications with standard interface for monitoring performance metrics, the harvester can simply leverage it. Otherwise, the harvester uses the swapped-in page count (promotion rate) as a proxy for performance [202]. Besides, other Linux features (e.g., Pressure-Stall Information [**?**]) also provide insights on an application's resource usage. One can leverage those information and correlate them to the application's performance metric. All of these approaches are transparent to the applications – Memtrade doesn't need to modify the application to monitor its performance.

Estimating the Baseline. To determine whether the memory limit should be decreased or increased, the harvester compares the current application performance metric to baseline values observed *without memory harvesting*. Of course, measuring performance without memory harvesting is difficult while the producer is actively reclaiming memory. To estimate the baseline performance without harvesting, we use statistics for swap-in events. When there are no swap-in events, the application has enough memory to run its workload. Therefore, the harvester includes the performance metric collected during these times as a baseline. An efficient AVL-tree data structure is used to track these points, which are discarded after an expiration time. Our current implementation adds a new data point every second, which expires after a 6-hour `WindowSize`. We found this yielded good performance estimates; shorter data-collection intervals or longer expiration times could further improve estimates, at the cost of higher resource consumption (§4.7.1).

Detecting Performance Drops. To decide if it can safely reduce the cgroup memory limit, the harvester checks whether performance has degraded more than expected from its estimated baseline performance. Similar to baseline estimation, the harvester maintains another AVL tree to track



(a) **Zipfian on Redis** (b) **XGBoost**

Figure 4.6: Performance degradation caused by harvesting different amounts of memory with and without Silo for (a) Zipfian trace running on Redis, and (b) XGBoost training.

application performance values over the same period.

After each performance-monitoring epoch, it calculates the 99th percentile (p99) of the recent performance distribution. The harvester assumes performance has dropped if the recent p99 is worse than baseline p99 by `P99Threshold` (by default, 1%), and it stops reducing the cgroup size, entering a recovery state. It then releases harvested memory adaptively to minimize the performance drop. Different percentiles or additional criteria can be used to detect performance drops.

Effective Harvesting with Silo. The harvester reclaims memory until a performance drop is detected. However, some workloads are extremely sensitive, and losing even a small amount of hot memory can result in severe performance degradation. Also, because the harvester adjusts the application memory size via a cgroup, it relies on the Linux kernel's Page Frame Reclamation Algorithm (PFRA) to make decisions. Unfortunately, PFRA is not perfect and sometimes reclaims hot pages, even with an appropriate memory limit.

To address these problems, we design *Silo*, a novel in-memory area for temporarily storing swapped-out pages. We implement Silo as a loadable kernel module that is a backend for the Linux frontswap interface [37]. The guest kernel swaps pages to Silo instead of disk, thus reducing the cost of swapping (Figure 4.5a). If a page in Silo is not accessed for a configurable `CoolingPeriod` (by default, 5 minutes), it is evicted to a slower memory tier (which is disk in a traditional server setup). Otherwise, an access causes it to be efficiently mapped back into the application's address space (Figure 4.5b). In effect, Silo is an in-memory victim cache, preventing hot pages from being swapped to slow memory tier. Figure 4.6 shows that Silo can prevent performance cliffs, allowing the harvester to avoid significant performance degradation.

When the harvester initially causes some pages to be swapped out to Silo, it cannot determine

the performance impact until the `CoolingPeriod` ends. As a result, it harvests cautiously, monitoring the application's RSS, which is available in the memory stats for its cgroup. If it enters harvesting mode, triggering the PFRA to move some pages to Silo, then the harvester refrains from further decreasing the cgroup limit for at least the `CoolingPeriod`. Afterwards, if performance is still stable, the harvester may resume decreasing the limit. This ensures that the harvester will not explore the harvesting size too aggressively without considering the performance impact caused by any disk I/O.

Handling Workload Bursts. Simply disabling the cgroup memory limit may not prevent performance drops in the face of sudden bursts. Memtrade addresses this issue by prefetching previously-reclaimed pages, proactively swapping them in from slow memory tiers. If the current performance is worse than all the recorded baseline data points for consecutive epochs, the harvester instructs Silo to prefetch `ChunkSize` of the most recently swapped-out pages (Figure 4.5c). Producers with a low tolerance for performance degradation and compressible data could alternatively use a compressed RAM disk [87] instead of a disk-based swap device. This would provide more rapid recovery, trading off total harvestable memory.

4.4.2 Exposing Remote Memory to Consumers

The manager communicates with the broker to report resource availability, and it exposes a KV or swap interface to consumers. The harvested memory space is logically partitioned into fixed-size slabs; a slab (by default, 64 MB) is the granularity at which memory is leased to consumers. Different slabs from the same producer can be mapped to multiple consumers for performance and load balancing. Upon receiving a message from the broker, the manager create a lightweight producer store in the producer VM, dedicated to serving remote memory for that consumer.

In Memtrade, the producer store is implemented by running a Redis [67] server within a cgroup in the producer VM, providing a familiar KV cache interface to consumers. Since an empty Redis server consumes only 3 MB of memory and negligible CPU, for simplicity, the manager runs a separate producer store for each consumer. The producer can limit the maximum CPU used by producer stores via cgroup controls. However, producer-side CPU consumption is typically modest; YCSB on Redis uses 3.1% of a core on average.

When the lease period expires, before terminating the Redis server, the manager checks with the broker to determine if the consumer wants to extend its lease (at the current market price). Otherwise, the producer store is terminated and slabs are returned to the remote memory pool.

Network Rate Limiter. The manager limits the amount of network bandwidth used by each consumer. We implemented a standard token-bucket algorithm [187] to limit consumer band-

width. The manager periodically adds tokens to each consumer bucket, in proportion to its allotted bandwidth specified in the consumer request for remote memory. Before serving a request, the producer store checks the consumer's available token count; if the I/O size exceeds the number of available tokens, it refuses to execute the request and notifies the consumer.

Eviction. The size of a producer store is determined by the amount of memory leased by each consumer. Once a producer store is full, it uses the default Redis eviction policy, a probabilistic LRU approximation [255]. In case of sudden memory bursts, the manager must release memory back to the producer rapidly. In this scenario, the harvester asks the manager to reclaim an aggregate amount of remote memory allocated to consumers. The manager then generates per-consumer eviction requests proportional to their corresponding producer store sizes and employs the approximate-LRU-based eviction for each producer store.

Defragmentation. The size of KV pairs may be smaller than the OS page size [242, 126, 114], which means that an application-level eviction will not necessarily free up the underlying OS page if other data in the same page has not been evicted. Fortunately, Redis supports memory defragmentation, which the producer store uses to compact memory.

4.5 Broker

The Memtrade broker is a trusted third-party that facilitates transactions between producers and consumers. It can be operated by the cloud provider, or by another company that runs the market as a service, similar to existing caching-as-a-service providers [57, 68]. Producers and consumers participate in the remote-memory market voluntarily, by registering their respective credentials with the broker. Each producer periodically sends its resource utilization metrics to the broker, which uses the resulting historical time series to predict future remote-memory availability over requested lease periods. Consumers request remote memory by sending allocation requests to the broker with the desired number of slabs and lease time, along with other preferences such as acceptable latency and bandwidth. The broker connects producers and consumers that may reside in separate virtual private clouds (VPCs) via existing high-bandwidth peering services [85, 39]. The broker maps consumer requests to producer slabs using an assignment algorithm that satisfies consumer preferences, while minimizing producer overhead and ensuring system-wide wellness objectives (e.g., load balancing and utilization).

In our current implementation, the broker runs on a single node and can handle a market with thousands of participating VMs (§4.7.2). Since consumers communicate directly with assigned producers until their leases expire, even if the broker is temporarily unavailable, the system can still

continue to operate normally, except for the allocation of new remote memory. For higher availability, the broker state could be replicated using distributed consensus, e.g., leveraging Raft [247, 33] or ZooKeeper [172]. The Memtrade operator may also run several broker instances, each serving a disjoint set of consumers and producers (e.g., one broker per region or datacenter).

4.5.1 Availability Predictor

Remote memory is transient by nature and can be evicted at any time to protect the performance of producer applications. Hence, allocating remote memory without considering its availability may result in frequent evictions that degrade consumer performance. Fortunately, application memory usage often follows a predictable long-term pattern, such as exhibiting diurnal fluctuations [138]. The broker capitalizes on historical time series data for producer memory consumption, predicting the availability of offered remote memory using an Auto Regressive Integrated Moving Average (ARIMA) model [168]. Producers with completely unpredictable usage patterns are not suitable for Memtrade. ARIMA model parameters are tuned daily via a grid search over a hyperparameter space to minimize the mean squared error of the prediction.

4.5.2 Remote Memory Allocation

Constraints and Assumptions. While matching a consumer's remote memory request, the broker tries to achieve the aforementioned goals under the following assumptions:

1. *Online requests:* Consumers submit remote memory requests in an online manner. During a placement decision, new or pending requests may be queued.

2. *Uncertain availability:* It is unknown exactly how long producer remote memory slabs will remain available.

3. *Partial allocation:* The broker may allocate fewer slabs than requested, as long as it satisfies the minimum amount specified by the consumer.

Placement Algorithm. When the broker receives an allocation request from a consumer, it checks whether at least one producer is expected to have at least one slab available for the entire lease duration (§4.5.1), at a price that would not exceed the consumer budget (§4.5.3). The broker calculates the placement cost of the requested slabs based on the current state of all potential producers with availability, as a weighted sum of the following metrics: number of slabs available at a given producer, predicted availability (based on ARIMA modeling), available bandwidth and

CPU, network latency between the consumer and producer, and producer reputation (fraction of remote memory not prematurely evicted during past lease periods – premature eviction of consumer data hurts a producer's reputation). A consumer may optionally specify weights for each of these placement desirability metrics with its request.

The broker selects the producer with the lowest placement cost, greedily assigning the desired number of slabs. If the producer cannot allocate the entire request, the broker selects the producer with the next-lowest cost and continues iteratively until there are no slabs left to allocate or no available producers. When fewer than the requested number are allocated, a request for the remaining slabs is appended to a queue. Pending requests are serviced in FIFO order until they are satisfied, or they are discarded after a specified timeout.

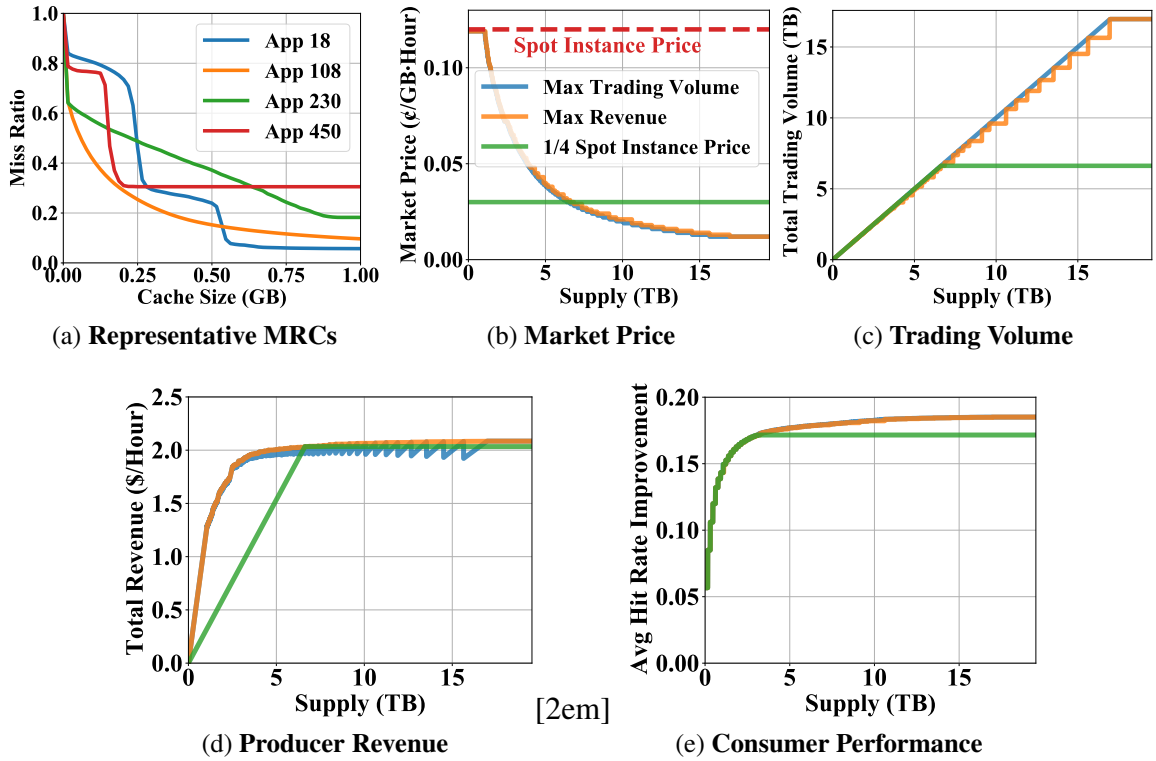
4.5.3 Remote Memory Pricing

Remote memory must be offered at a price that is attractive to both producers and consumers, providing incentives to participate in the remote memory market. Considering the transient nature of harvested memory, any monetary incentive for leasing otherwise-wasted resources is beneficial to a producer, provided its own application-level performance is not impacted. This incentive can help the producer defray the expense of running its VM. A consumer must weigh the monetary cost of leasing remote memory against the cost of running a static or spot instance with larger memory capacity. In case if a rack-scale tiered memory system within any private cloud, the goal for the broker can be better resource utilization while maintaining the service-level agreement (SLA) for each applications.

In a public cloud setup, the broker sets a price for leasing a unit of remote memory (GB/hour) and makes it visible to all consumers. Various economic objectives could be optimized (e.g., total trading volume, total revenue of producers, etc.) If the primary goal is improving cluster-wide utilization, the broker needs to focus on maximizing the total trading volume that can be achieved by setting a market-clearing price for remote memory. In this regards, the broker can continuously monitor the price of different instance types in the same public cloud. Specifically, for a given lease, the broker can set the per-unit remote memory price to a value below that associated with the lowest available instance price at that time. On the other hand, if the broker wants to maximize its cut of the revenue, it will try to optimize the total revenue of producers. By default, we assume, the broker wants to maximize its profit. Although, one can come up with any complex market dynamics and plug in their customized policies to achieve the broker's goal or market economics as a whole. Memtrade provides the interface to operate on any pricing model to address different economic objectives. It's up to its users to choose the appropriate policy.

From a consumer's perspective, an alternative to Memtrade is running a separate spot instance

and consuming its memory remotely [296]. Thus, to be economically viable to consumers, the price of remote memory in Memtrade should never exceed the corresponding spot instance price. For simplicity, the broker initially sets the price for each unit of remote memory to one quarter of the current market price for a spot instance, normalized by its size. This initial lower price makes remote memory attractive to consumers. Afterwards, the price is adjusted to approximate the maximal total producer revenue by searching for a better price locally. In each iteration, the broker considers the current market price p, $p + \Delta p$, and $p - \Delta p$ as the candidates for the price in the next iteration, where Δp is the step size (by default, 0.002 cent/GB·hour). Then the broker chooses the one that generates the maximal total revenue for producers. Our pricing model yields good performance in real-world traces (§4.5.4). Of course, alternative price-adjustment mechanisms can be designed to achieve different economic objectives. One can simply add on their own pricing model and economic objectives to the broker and enjoy the benefit of Memtrade.



(a) **Representative MRCs** (b) **Market Price** (c) **Trading Volume**

[2em]

(d) **Producer Revenue** (e) **Consumer Performance**

Figure 4.7: Effects of different pricing strategies. Revenue and trading-volume maximization both perform similarly.

4.5.4 Pricing Strategy

To study the impact of pricing strategy on the market, we simulate several pricing strategies with different objectives, such as maximizing the total trading volume, and maximizing the total revenue of producers. Our baseline sets the remote memory price to one quarter of the current spot-instance price (Figure 4.7b). We simulate 10,000 consumers that use Memtrade as an inexpensive cache. To estimate the expected performance benefit for consumers, we select 36 applications from the MemCachier trace, generate their MRCs (Figure 4.8), and randomly assign one to each consumer. For each consumer, we ensure that local memory serves at least 80% of its optimal hit ratio.

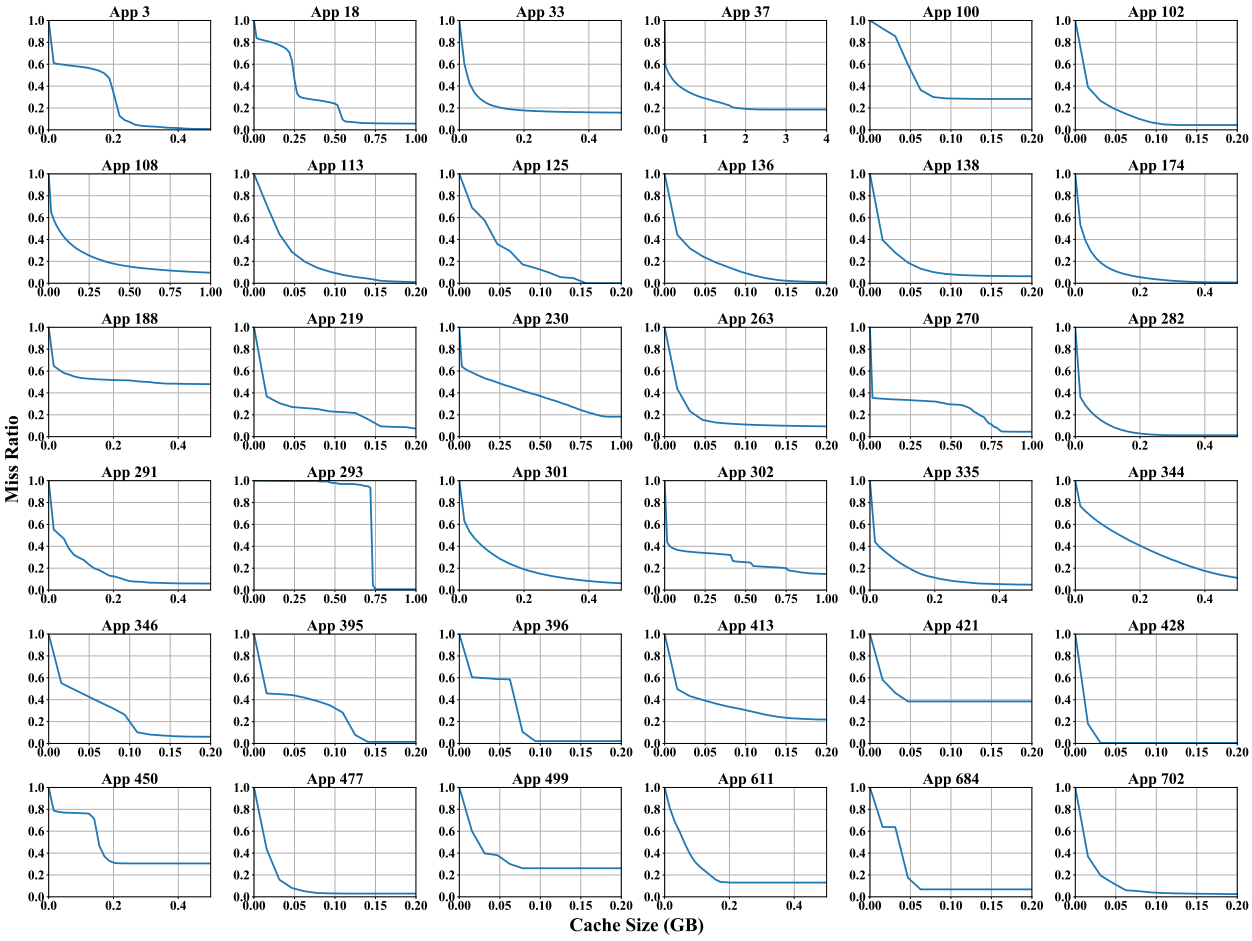
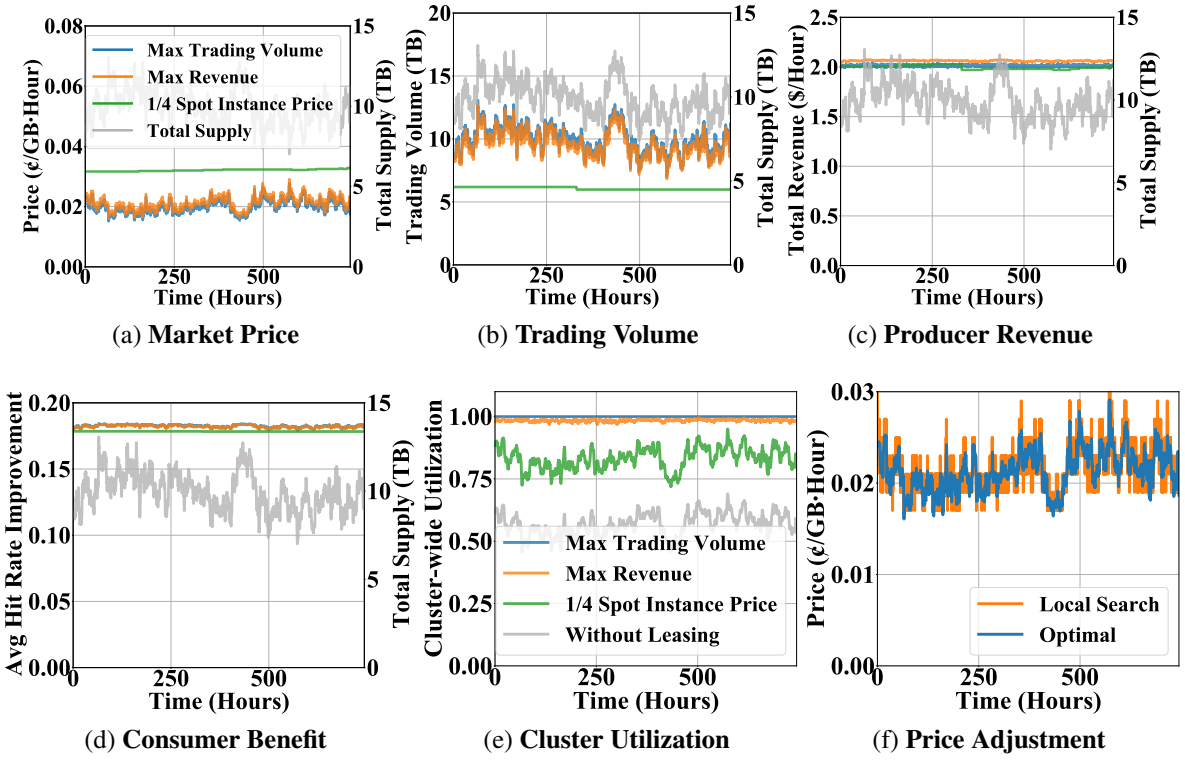


Figure 4.8: Miss Ratio Curves of 36 MemCachier Applications

The strategies that maximize total trading volume (Figure 4.7c) and total producer revenue (Figure 4.7d) both adjust the market price dynamically to optimize their objectives. Significantly, when the remote-memory supply is sufficient, all three pricing strategies can improve the relative hit ratio for consumers by more than 16% on average (Figure 4.7e).

We also examine temporal market dynamics using a real-world trace to simulate a total supply

of remote memory which varies over time. We use the idle memory statistics from the Google Cluster Trace 2019 – Cell C [41] to generate the total supply for each time slot, and assume one Google unit represents 5 GB of memory. For the spot instance price, we use the AWS historical price series of the spot instance type r3.large in the region us-east-2b [17]. Figure 4.9 plots the results. Consistent with the earlier pricing-strategy results in Figure 4.7, the max-trading-volume and max-revenue strategies effectively adjust the market price based on both supply and demand (Figure 4.9a). The behavior of all the three pricing strategies with different economic objectives show similar levels of consistency (Figure 4.9b–4.9e).



(a) **Market Price** (b) **Trading Volume** (c) **Producer Revenue**

(d) **Consumer Benefit** (e) **Cluster Utilization** (f) **Price Adjustment**

Figure 4.9: Temporal market dynamics with simulated supply time-series from Google Cluster Trace 2019.

We also consider a more realistic scenario where consumers consider the probability of being evicted when using MRCs to calculate their demand. If the eviction probability is 10%, the total revenue will decrease by 7.6% and 7.1% with the max-trading-volume strategy and the max-revenue strategy, respectively. Also, the cluster-wide utilization reductions corresponding to the max-trading-volume strategy and the max-revenue strategy are 0.0% and 5.8% relatively.

In practice, the broker may have no prior knowledge regarding the impact of market price on consumer demand. In this case, we adjust the market price by searching for a better price locally with a predefined step size (0.002 cent/GB·hour). Figure 4.9f demonstrates the effectiveness of

this approach using a simulation with the Google trace; the market price deviates from the optimal one by only 3.5% on average. Cluster-wide utilization increases from 56.8% to 97.9%, consumer hit ratios improve by a relative 18.2%, and the consumer's cost of renting extra memory reduces by an average of 82.1% compared to using spot instances.

4.6 Consumer

A consumer uses remote memory. It first sends its remote memory demand to the broker, based on the current market price given by the broker and its expected performance benefit. After receiving an assignment from the broker, the consumer communicates with producers directly during the lease period. To ensure the confidentiality and integrity of its remote data, the consumer employs standard cryptographic methods during this communication. Rate-limiting techniques are used to protect producers from misbehaving or malicious consumers.

The consumer can use either a KV cache or a swap interface, which we built on top of Redis [67] and Infiniswap [165] clients, respectively. By default, Memtrade uses the key-value interface, because in contrast to swapping to disk, applications using a KV cache naturally assume cached data can disappear. We have also found the KV interface performs better than the swap interface (§4.7.3), due to the added overhead of going through the block layer when swapping. For the sake of brevity, we focus our description on the KV interface.

4.6.1 Confidentiality and Integrity

This section explains how the consumer ensures data confidentiality and integrity during its KV operations. The subscripts C and P are used to denote consumer-visible and producer-visible data, respectively.

PUT Operations. To perform a `PUT`, the consumer prepares a KV pair (K_C, V_C) to be stored at a remote producer. First, the value V_C is encrypted using the consumer's secret key and a fresh, randomly-generated initialization vector (IV). The IV is prepended to the resulting ciphertext, yielding the value V_P to be stored at the producer. Next, a secure hash H is generated for V_P, to verify its integrity and defend against accidental or malicious corruption by the producer.

To avoid exposing the lookup key K_C, the consumer substitutes a different key K_P. Since K_P need only be unique, it can be generated efficiently by simply incrementing a counter for each new key stored at a producer. The producer store storing the KV pair can be identified using an index P_i into a small table containing producer information.

The consumer stores the metadata tuple M_C = (K_P, H, P_i) locally, associating it with K_C. While many implementations are possible, this can be accomplished conveniently by adding (K_C, M_C) to a local KV store, where an entry serves as a proxy for obtaining the corresponding original value. This approach also enables range queries, as all original keys are local.

GET Operations. To perform a `GET`, the consumer first performs a local lookup using K_C to retrieve its associated metadata M_C, and sends a request to the producer using substitute key K_P. The consumer verifies that the value V_P returned by the producer has the correct hash H; if verification fails, the corrupted value is discarded. The value V_P is then decrypted using IV with the consumer's encryption key, yielding V_C.

DELETE Operations. To perform a consumer-side eviction, the consumer first removes the metadata tuple M_C from its local store. It then sends an explicit `DELETE` request to the respective producer store so that the consumer and producer store contents remain synchronized.

Metadata Overhead. In our current prototype, each consumer uses a single secret key to encrypt all values. Encryption uses AES-128 in CBC mode, and hashing uses SHA-256, both standard constructions. By default, the integrity hash is truncated to 128 bits to save space. A 64-bit counter is employed to generate compact producer lookup keys. The resulting space overhead for the metadata M_C corresponding to a single KV pair (K_C, V_C) is 24 bytes; the IV consumes an additional 16 bytes at the producer.

For applications where consumer data is not sensitive, value encryption and key substitution are unnecessary. Such an integrity-only mode requires only the integrity hash, reducing the metadata overhead to 16 bytes.

4.6.2 Purchasing Strategy

A consumer must determine a cost-effective amount of memory to lease to meet its application-level performance goals. In general, it may be difficult to estimate the monetary value of additional memory. However, when its application is a cache, lightweight sampling-based techniques [295, 170, 293] can estimate miss ratio curves (MRCs) accurately, yielding the expected performance benefit from a larger cache size.

The consumer estimates the value of additional cache space using the current *price-per-hit* from the known cost of running its VM, and its observed hit rate. The expected increase in hits is computed from its MRC, and valued based on the per-hit price. When remote memory is more valuable to the consumer than its cost at the current market price, it should be leased, yielding an economic consumer surplus.

	Total Harvested	Idle Harvested	Workload Harvested	Perf Loss
Redis	3.8 GB	23.7%	17.4%	0.0%
memcached	8.0 GB	51.4%	14.6%	1.1%
MySQL	4.2 GB	21.7%	7.0%	1.6%
XGBoost	18.3 GB	15.4%	17.8%	0.3%
Storm	3.8 GB	1.1%	1.4%	0.0%
CloudSuite	3.6 GB	2.5%	15.3%	0.0%

Table 4.1: Total memory harvested (idle and unallocated), the percentage of idle memory, the percentage of application-allocated memory, and the performance loss of different workloads.

4.7 Evaluation

We evaluate Memtrade on a CloudLab [25] cluster using both synthetic and real-world cluster traces.[1] Our evaluation addresses the following questions:

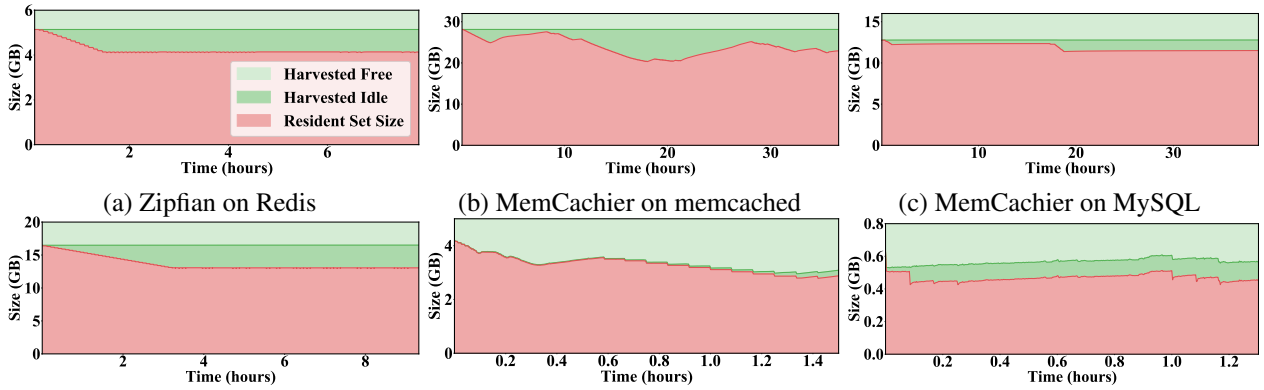
- How effectively can memory be harvested? (§4.7.1)
- How well does the broker assign remote memory? (§4.7.2)
- What are Memtrade's end-to-end benefits? (§4.7.3)

Experimental Setup. Unless otherwise specified, we configure Memtrade as follows. The producer averages application-level latency over each second as its performance metric. We generate both the baseline performance distribution and the recent performance distribution from data points over the previous 6 hours (`WindowSize`). If the recent p99 drops below the baseline p99 by more than 1% (`P99Threshold`), it is considered a performance drop. Harvesting uses a 64 MB `ChunkSize` and a 5-minute `CoolingPeriod`. If a severe performance drop occurs for 3 consecutive epochs, Silo prefetches `ChunkSize` from disk.

Each physical server is configured with 192 GB DRAM, two Intel Xeon Silver 4114 processors with 20 cores (40 hyperthreads), and a 10Gb NIC. We use Intel DC S3520 SSDs and 7200 RPM SAS HDDs. We run the Xen hypervisor (v4.9.2) with Ubuntu 18.04 (kernel v4.15) as the guest OS.

Workloads. Consumers run YCSB [137] on Redis [67]. Producers run the following applications and workloads:

[1] Memtrade can be readily deployed on any major cloud provider. We run our evaluation in CloudLab since it is free.



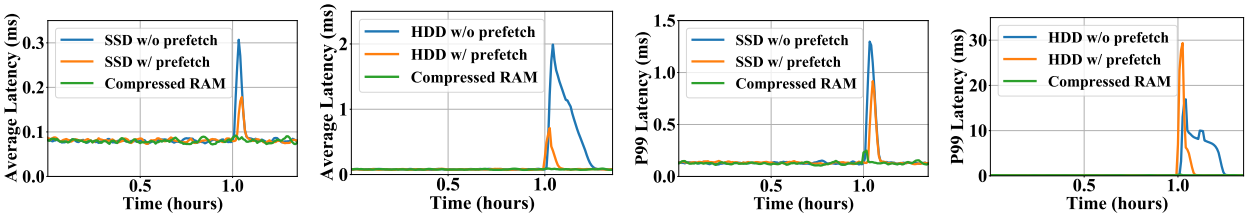
(a) Zipfian on Redis (b) MemCachier on memcached (c) MemCachier on MySQL

(d) Training classifier on XGBoost (e) Yahoo Streaming on Apache Storm (f) CloudSuite Web Serving

Figure 4.10: VM memory composition over time. Unallocated represents the part of memory not allocated to the application; harvested means the portion of application's memory which has been swapped to disk; Silo denotes the part of memory used by Silo to buffer reclaimed pages; RSS consists of application's anonymous pages, mapped files, and page cache that are collected from the cgroup's stats file.

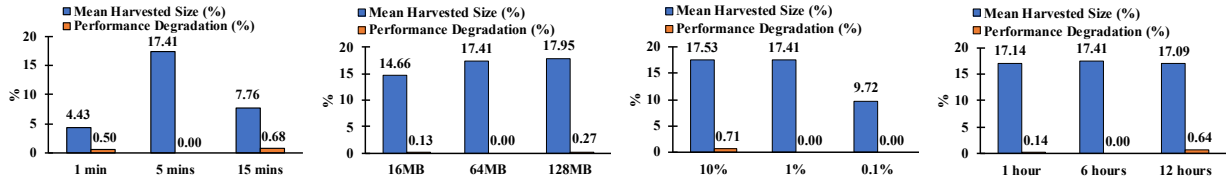
- **Redis** running a Zipfian workload using a Zipfian constant of 0.7 with 95% reads and 5% updates.
- **memcached** and **MySQL** running MemCachier [57, 125] for 36 hours and 40 hours, respectively. We use 70 million `SET` operations to populate the memcached server, followed by 677 million queries for memcached and 135 million queries for MySQL.
- **XGBoost** [121] training an image classification model on images of cats and dogs [24] using CPU, with 500 steps.
- **Storm** [72] running the Yahoo streaming workload [124] for 1.5 hours.
- **CloudSuite** [43] executing a web-serving benchmark with memcached as the cache and MySQL as the database, with 1000 users and 200 threads.

VM Rightsizing. To detemine the VM size for each workload, we find the AWS instance type [6] with the minimal number of cores and memory that can fit the workload without affecting its baseline performance. We use configurations of M5n.Large (2 vCPU, 8 GB RAM) for Redis, M5n.2xLarge (8 vCPU, 32 GB RAM) for memcached and XGBoost, C6g.2xLarge (8 vCPU, 16 GB RAM) for MySQL, C6g.xLarge (4 vCPU, 8 GB RAM) for Storm, C6g.Large (2 vCPU, 4 GB RAM) for CloudSuite, and T2.xLarge (4 vCPU, 16 GB RAM) for consumer YCSB.



(a) **SSD Swap Device** (b) **HDD Swap Device** (c) **SSD Swap Device** (d) **HDD Swap Device**

Figure 4.11: Prefetching enables faster recovery and better latency during workload bursts. Memory compression enables even faster recovery, trading off the total amount of harvestable memory.



(a) **Silo `CoolingPeriod`** (b) **Harvesting ChunkSize** (c) **`P99Threshold`** (d) **Monitor `WindowSize`**

Figure 4.12: **Sensitivity analysis for the harvester.** Single-machine experiments using Redis with YCSB Zipfian constant 0.7.

4.7.1 Harvester

Effectiveness. To observe the effectiveness of the harvester, we run the workloads with their respective producer configurations. For Redis, memcached, and MySQL we use average latency to measure performance. Since XGBoost, Storm, and Cloudsuite do not provide any real-time performance metric, we use the promotion rate (number of swapped-in pages) as a proxy for performance.

We find that Memtrade can harvest significant amounts of memory, even from right-sized VMs (Table 4.1). Here, a notable portion of the total harvested memory is extracted from the application's idle memory (on average, 1.1–51.4% across the entire workload) at a lower performance degradation cost of 0–1.6%. Also, a whole-machine, all-core analysis shows that the producer-side CPU and memory overheads due to the harvester were always less than 1%.

Figure 4.10 plots memory allocation over time for two representative workloads, and shows that for workloads such as MemCachier with varying access patterns (Figure 4.10b), the harvester dynamically adjusts the amount of harvested memory. For most workloads, the percentage of idle harvested memory is higher at the end of the run. Therefore, we expect that if we ran our workloads longer, the average percentage of idle harvested memory would only increase.

Impact of Burst-Mitigation Techniques. To observe the effectiveness of the harvester during workload bursts, we run YCSB on Redis using a Zipfian distribution (with constant 0.7). To create a workload burst, we abruptly shift it to a uniform distribution after one hour of the run. Figure 4.11 shows the average latency and p99 latency using different burst-mitigation approaches. When enabled, Silo prefetches harvested pages back into memory, which helps the application reduce its recovery time by 28.6% and 65.5% over SSD and HDD, respectively. Although prefetching causes a higher peak p99 latency on HDD due to contending with swapping I/O, this high tail latency is transient and the application can recover much faster. A compressed RAM disk exhibits minimal performance drop period during the workload burst (68.7% and 10.9% less recovery time over prefetching from SSD and HDD, respectively), at the cost of less harvested memory.

Sensitivity Analysis. We run YCSB over Redis with a Zipfian constant of 0.7 to understand the effect of each parameter on harvesting and producer performance. Each parameter is evaluated in an isolated experiment. Figure 4.12 reports the results, using average performance to quantify degradation.

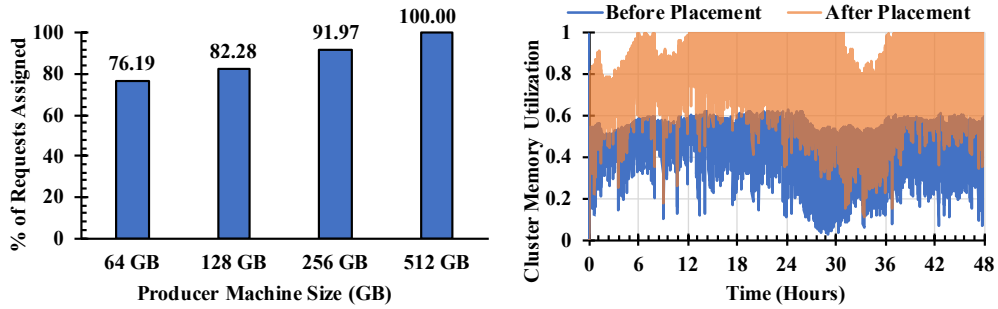
The Silo `CoolingPeriod` controls the aggressiveness of harvesting (Figure 4.12a). Setting it too high leads to less harvesting, while setting it too low causes performance drops that eventually also leads to less harvested memory.

Both the harvesting `ChunkSize` (Figure 4.12b) and the `P99Threshold` (Figure 4.12c) affect harvesting aggressiveness in a less pronounced way. The amount of harvested memory increases with more aggressive parameters, while the performance impact always remains below 1%. The performance-monitoring `WindowSize` does not significantly change either the harvested memory or performance (Figure 4.12d).

4.7.2 Broker Effectiveness

To evaluate the effectiveness of the broker, we simulate a disaggregated memory consumption scenario by replaying two days worth of traces from Google's production cluster [41]. Machines with high memory demand – often exceeding the machine's capacity – are treated as consumers. Machines with medium-level memory pressure (at least 40% memory consumption throughout the trace period) are marked as producers. When a consumer's demand exceeds its memory capacity, we generate a remote memory request to the broker. We set the consumer memory capacity to 512 GB, the minimum remote memory slab size to 1 GB, and a minimum lease time of 10 minutes. In our simulation, 1400 consumers generate a total of 10.7 TB of remote memory requests within 48 hours. On the producer side, we simulate 100 machines.

On average, the broker needs to assign 18 GB of remote memory to the producers per minute.



(a) **Placement** (b) **Cluster-wide Utilization**

Figure 4.13: Simulation of remote memory usage shows the broker allocates most requests and improves cluster-wide utilization.

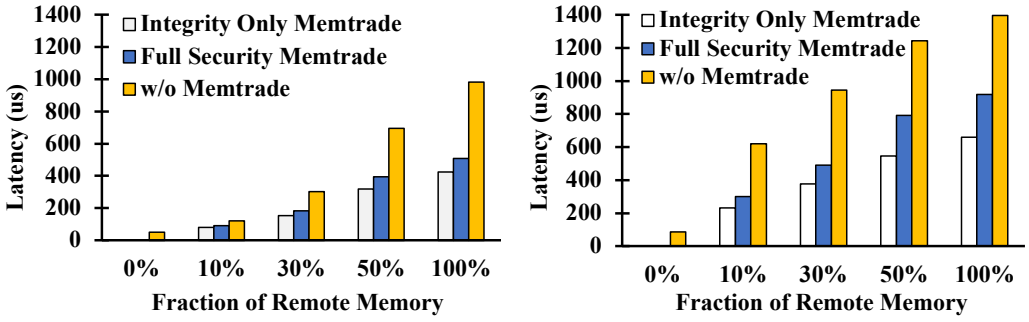
Our greedy placement algorithm can effectively place most requests. Even for a simulation where producers have only 64 GB DRAM, it can satisfy 76% of the requests (Figure 4.13a). With larger producers, the total number of allocations also increases. As expected, Memtrade increases cluster-wide utilization, by 38% (Figure 4.13b).

Availability Predictor. To estimate the availability of producer memory, we consider its average memory usage over the past five minutes to predict the next five minutes. ARIMA predictions are accurate when a producer has steady usage or follows some pattern; only 9% of the predicted usage exceeds the actual usage by 4%. On average, 4.59% of the allocated producer slabs get revoked before their lease expires.

4.7.3 Memtrade's Overall Impact

Encryption and Integrity Overheads. To measure the overhead of Memtrade, we run the YCSB workload over Redis with 50% remote memory. Integrity hashing increases remote memory access latency by 24.3% (22.9%) at the median (p99). Hashing and key replacement cause 0.9% memory overhead for the consumer. Due to fragmentation in the producer VM, the consumer needs to consume 16.7% extra memory over the total size of actual KV pairs. Note that this fragmentation overhead would be same if the consumer had to store the KVs in its local memory.

Encryption and key substitution increase latency by another 19.8% (14.7%) at the median (p99). Due to padding, encryption increases the memory overhead by another 25.2%. Fragmentation in the producer VM causes 6.1% memory overhead. Note that for trusted producers, or for non-sensitive consumer data, encryption can be disabled.



(a) **Average Latency** (b) **p99 Latency**

Figure 4.14: Benefit of Memtrade with various configurations. Without Memtrade, remote requests are served from SSD.

Application-Level Performance. We run YCSB over Redis with different consumer VM configurations and security modes. The consumer memory size is configured such that Redis needs at least x% ($x \in \{0, 10, 30, 50, 100\}$) of its working set to be in remote Memtrade memory. If remote memory is not available, the I/O operation is performed using SSD.

For the 0% configuration, the entire workload fits in the consumer VM's memory, and is fully served by its local Redis. Figure 4.14 shows that an application benefits from using remote memory as additional cache instead of missing to disk. For the fully-secure KV cache interface, Memtrade improves average latency by 1.3–1.9×, and p99 latency by 1.5–2.1×. In the case of non-sensitive data with only integrity checks, Memtrade provides 1.45–2.3× and 2.1–2.7× better average and p99 latencies, respectively.

We also implemented a transparent swap-based consumer interface, built on top of Infiniswap [165]. When consuming remote memory via fully-secure remote paging, Memtrade's performance drops due to hypervisor swapping overhead – average and p99 latency drop by 0.95–2.1× and 1.1–3.9×, respectively. However, given a faster swapping mechanism [224, 267] or faster network (e.g., RDMA), Memtrade swap is likely to provide a performance benefit to consumers.

Cluster Deployment. To measure the end-to-end benefit of Memtrade, we run 110 applications on CloudLab – 64 producer and 46 consumer VMs, randomly distributed across the cluster. Producers run six workloads described earlier, while consumers run YCSB on Redis, configured with 10%, 30%, and 50% remote memory. The total consumer and producer memory footprints are 0.7 TB and 1.3 TB, respectively; our cluster has a total of 2.6 TB. Memtrade benefits the consumers even in a cluster setting (Table 4.2). Memtrade improves the average latency of consumer applications by 1.6–2.8×, while degrading the average producer latency by 0.0–2.1%.

Producer Application	Avg. Latency (ms)		Consumer Application	Avg. Latency (ms)	
	w/o Harvester	w/ Harvester		w/o Memtrade	w/ Memtrade
Redis	0.08	0.08	Redis 0%	0.62	–
Memcached	0.82	0.83	Redis 10%	1.10	0.71
MySQL	1.57	1.60	Redis 30%	1.54	0.88
Storm	5.33	5.47	Redis 50%	2.49	0.89

Table 4.2: Memtrade benefits consumers at a small cost to producers.

4.8 Discussion

Network Stack. As public cloud environments are typically virtualized, inter-VM networking incurs additional latency due to traversing both kernel and hypervisor networking stacks. As shown in Figure 4.15, kernel-bypass networking (e.g., DPDK, RDMA) offers lower latency in virtualized environments; with RDMA and SR-IOV, the additional overhead is minimal compared to bare-metal machines. Unfortunately, RDMA is not widely available in existing public clouds. While DPDK is slower than RDMA, it still achieves single-digit μs latencies for small objects. However, it consumes an excessive amount of CPU due to polling (nearly 100% of local CPU for larger objects), which would require producers to have extremely idle CPU resources to be practical.

CXL [27] is a new processor-to-peripheral/accelerator cache-coherent interconnect protocol that builds on and extends the existing PCIe protocol by allowing coherent communication between the connected devices.[2] It provides byte-addressable memory in the same physical address space and allows transparent memory allocation using standard memory allocation APIs. It also allows cache-line granularity access to the connected devices and underlying hardware maintains cache-coherency and consistency. With PCIe 5.0, CPU-to-CXL interconnect bandwidth is similar to the cross-socket interconnects on a dual-socket machine [326]. CXL-Memory access latency is also similar to the NUMA access

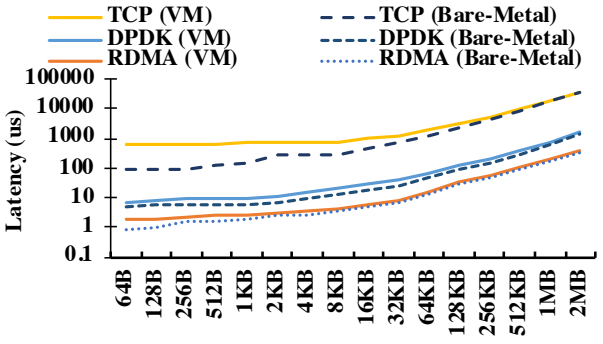


Figure 4.15: Network latency using VMs vs. bare-metal hosts for various protocols and different object sizes.

[2]Prior industry standards in this space such as CCIX [22], OpenCAPI [62], Gen-Z [38] etc. have all come together under the banner of CXL consortium. While there are some related research proposals (e.g., [206]), CXL is the de facto industry standard at the time of writing this paper.

latency. CXL adds around 50-100 nanoseconds of extra latency over normal DRAM access.

Because of its universal availability, Memtrade currently uses TCP networking, despite its relatively high overhead and other limitations. For deployments in private clouds, replacing TCP with RDMA or even CXL-enabled rack-servers would improve performance significantly.

Consumer Abstractions. Memtrade is designed considering the readily deployability feature for most of the common usages in existing cloud infrastructure. To hold its ubiquitousness, we design the consumer APIs thinking TCPs as the most common network medium. Although it needs some re-write or modification to the consumer applications, on TCP networks, caching applications can usually sustain the network communication overhead. Swapping to remote memory can be a viable and transparent abstraction for consumer applications [224, 207, 111]. However, to maintain the performance while remote swapping, we need to ensure the network stack is at least as performant as RDMA over Infiniband. With faster and next-generation cache-coherent CXL interconnects, we are confident on Memtrade's applicability. In such a setup, we can design more transparent and performant interfaces to the consumers to consume remote memory efficiently. OS level features (i.e., migration-based page placement [225]) can be utilized to design new consumer APIs.

4.9 Related Work

Remote Memory. Existing work on *remote-memory-enabled* datacenter [275, 191, 166, 157, 202, 243, 214, 186, 94, 99, 92, 165, 267] assume that *memory is contained within the same organization and shared among multiple cooperative applications*. Given the large amount of idle memory and diverse consumer applications and workloads, public clouds serve as a promising environment to exploit remote memory.

Public Cloud Spot Marketplaces. Amazon AWS, Microsoft Azure, and Google Cloud offer spot instances [71] – a marketplace for unutilized public cloud VMs that have not been reserved, but have been provisioned by the cloud provider. AWS allows customers to bid on spot instances while Azure and Google Cloud [80, 63] sell them at globally fixed prices. Prior work explored the economic incentives for spot instances [89, 296, 327, 305, 282, 320, 108, 90, 156, 278, 159]. However, they used full spot instances to produce memory; Memtrade is more generic, enabling fine-grained consumption of excess memory from any instance type.

Single-Server Memory Management. Reclaiming idle memory from applications has been explored in many different contexts, e.g., physical memory allocation across processes by an OS [140, 130, 88], ballooning in hypervisors [294, 272, 277], transcendent memory for caching

disk swap [221, 222, 220], etc. Our harvesting approach is related to application-level ballooning [269]. However, in most prior work, applications belong to the same organization while Memtrade harvests and grants memory across tenants. Multi-tenant memory sharing [90, 156, 129] has considered only single-server settings, limiting datacenter-wide adoption.

Resource Autoscaling and Instance Rightsizing. Cloud orchestration frameworks and schedulers [290, 273, 52, 96, 13] can automatically increase or decrease the number of instances based on the task arrival rate. However, users still need to statically determine the instance size before launching a task, which may lead to resource overprovisioning.

Instance rightsizing [268, 82, 97, 100, 298] automatically determines the instance size based on a task's resource consumption. In existing solutions, cloud providers are fully responsible for resource reclamation and performance variability in producer VMs. Memtrade is by design more conservative: producers can generate and reclaim memory capacity at any time. Even with rightsizing, servers may have idle memory that can be harvested and offered to consumer VMs.

Memory Harvesting. Harvesting unused resources of a traditional server or VM is a concern in many hyperscale datacenters [155, 276]. Earlier research mostly focus on the unallocated memory of a VM as the harvesting target. However, besides that unallocated memory, there are significant amount of memory that has been allocated to an application; the application used that for a while; and then, remained unutilized for a significant amount of time [225, 301, 208, 202]. Memtrade harvester considers all kind of idle memory.

4.10 Conclusion

We present Memtrade, a readily deployable system for the realization of disaggregated memory marketplace on existing clouds. With the rising popularity of serverless and disaggregated storage, there will be increased demand for offering disaggregated computing resources in public clouds, and our work attempts to apply a similar approach to memory. This opens several interesting research directions for future work, including exploring whether other resources, such as CPU and persistent memory, can be offered in a similar manner via a disaggregated-resource market.

CHAPTER 5

TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory

5.1 Introduction

The surge in memory needs for datacenter applications [251, 34], combined with the increasing DRAM cost and technology scaling challenges [205, 219] has led to memory becoming a significant infrastructure expense in hyperscale datacenters. Non-DRAM memory technologies provide an opportunity to alleviate this problem by building tiered memory subsystems and adding higher memory capacity at a cheaper $/GB point [185, 146, 144, 21, 51]. These technologies, however, have much higher latency vs. main memory and can significantly degrade performance when data is inefficiently placed in different levels of the memory hierarchy. Additionally, prior knowledge of application behavior and careful application tuning is required to effectively use these technologies. This can be prohibitively resource-intensive in hyperscale environments with varieties of rapidly evolving applications.

Compute Express Link (CXL) [27] mitigates this problem by providing an intermediate latency operating point with DRAM-like bandwidth and cache-line granular access semantics. CXL protocol allows a new memory bus interface to attach memory to the CPU (Figure 5.1). From a software perspective, CXL-Memory appears to a system as a CPU-less NUMA node where its memory characteristics (e.g., bandwidth, capacity, generation, technology, etc.) are independent of the memory directly attached to the CPU. This allows flexibility in memory subsystem design and fine-grained control over the memory bandwidth and capacity [31, 30, 69]. Additionally, as CXL-Memory appears like the main memory, it provides opportunities for transparent page placement on the appropriate memory tier. However, Linux's memory management mechanism is designed for homogeneous CPU-attached DRAM-only systems and performs poorly on CXL-Memory system. In such a system, as memory access latency varies across memory tiers (Figure 5.2), application performance greatly depends on the fraction of memory served from the fast memory.

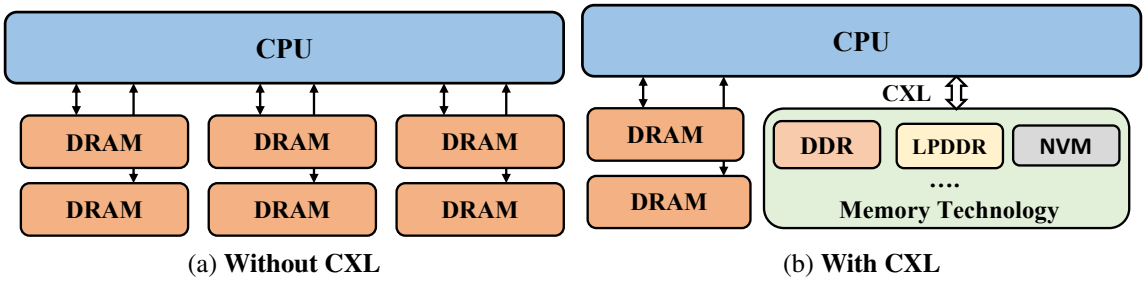


Figure 5.1: CXL decouples memory from compute.

To understand whether memory tiering can be beneficial, we need to understand the variety of memory access behavior in existing datacenter applications. For each application, we want to know how much of its memory remains hot, warm, and cold within a certain period and what fraction of its memory is short- vs. long-lived. Existing Idle Page Tracking (IPT) based characterization tools [32, 46, 291] do not fit the bill as they require kernel modifications that is often not possible in productions. Besides, continuous access bit sampling and profiling require excessive CPU and memory overhead. This may not scale with large working sets. Moreover, applications often have different sensitivity towards different types of memory pages (e.g., anon page, file page cache, shared memory, etc.) which existing tools do not account. To this end, we build Chameleon, a robust and lightweight user-space tool, that uses existing CPU's Precise Event-Based Sampling (PEBS) mechanism to characterize an application's memory access behavior (§5.3). Chameleon generates a heat-map of memory usage on different types of pages and provides insights into an application's expected performance with multiple temperature tiers.

We use Chameleon to profile a variety of large memory-bound applications across different service domains running in our production and make the following observations. **(1)** Meaningful portions of application working sets can be warm/cold. We can offload that to a slow tier memory without significant performance impact. **(2)** A large fraction of anon memory (created for a program's stack, heap, and/or calls to mmap) tends to be hotter, while a large fraction of file-backed memory tends to be relatively colder. **(3)** Page access patterns remain relatively stable for meaningful time durations (minutes to hours). This is enough to observe application behavior and make page placement decisions in kernel-space. **(4)** With new (de)allocations, actual physical page addresses can change their behavior from hot to cold and vice versa fairly quickly. Static page allocations can significantly degrade performance.

Considering the above observations, we design an OS-level transparent page placement mechanism – TPP, to efficiently place pages in a tiered-memory systems so that relatively hot pages remain in fast memory tier and cold pages are moved to the slow memory tier (§5.5). TPP has three prime components: **(a)** a lightweight reclamation mechanism to demote colder pages to the

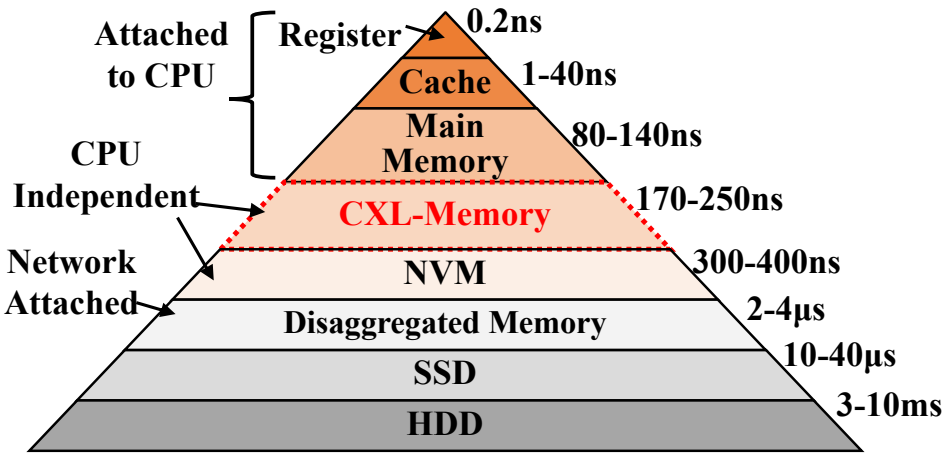


Figure 5.2: Latency characteristics of memory technologies.

slow tier node; **(b)** decoupling the allocation and reclamation logic for multi-NUMA systems to maintain a headroom of free pages on fast tier node; and **(c)** a reactive page promotion mechanism that efficiently identifies hot pages trapped in the slow memory tier and promote them to the fast memory tier to improve performance. We also introduce support for page type-aware allocation across the memory tiers – preferably allocate sensitive anon pages to fast tier and file caches to slow tier. With this optional application-aware setting, TPP can act from a better starting point and converge faster for applications with certain access behaviors.

We choose four production workloads that constitute significant portion of our server fleet and run them on a system that support CXL 1.1 specification (§5.6). We find that TPP provides the similar performance behavior of all memory served from the fast memory tier. For some workloads, this holds true even when local DRAM is only 20% of the total system memory. TPP moves all the effective hot memory to the fast memory tier and improves default Linux's performance by up to 18%. We compare TPP against NUMA Balancing [60] and AutoTiering [189], two state-of-the-art solutions for tiered memory. TPP outperforms both of them by 5–17%.

We make the following contributions in this paper:

- We present Chameleon, a lightweight user-space memory characterization tool. We use it to understand workload's memory consumption behavior and assess the scope of tiered-memory in hyperscale datacenters (§5.3).
- We propose TPP for efficient memory management on a tiered-memory system (§5.5). We publish the source code of TPP. A major portion of it has been merged to Linux kernel v5.18. Rest of it is under an upstream discussion.
- We evaluate TPP on a CXL-enabled tiered-memory systems with real production workloads

(§5.6) for years. TPP makes tiered memory systems as performant as an ideal system with all memory in local tier. For datacenter applications, TPP improves default Linux's performance by up to 18%. It also outperforms NUMA Balancing and AutoTiering by 5–17%.

To the best of our knowledge, we are the first to characterize and evaluate an end-to-end practical CXL-Memory system that can be readily deployed in hyperscale datacenters.

5.2 Motivation

Increased Memory Demand in Datacenter Applications. To build low-latency services, in-memory computation has become a norm in datacenter applications. This has led to rapid growth in memory demands across the server fleet. With new generations of CPU and DRAM technologies, memory is becoming the more prominent portion of rack-level power and TCO (Figure 5.3).

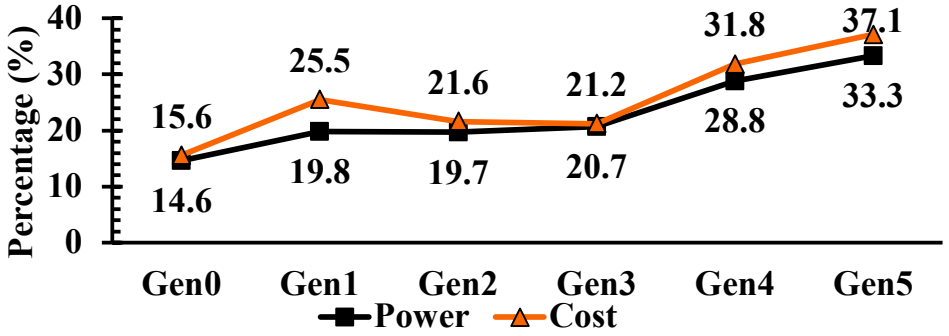


Figure 5.3: Memory as a percentage of rack TCO and power across Meta's different hardware generations.

Scaling Challenges in Homogeneous Server Designs. In today's server architectures, memory subsystem design is completely dependent on the underlying memory technology support in the CPUs. This has several limitations: (a) memory controllers only support a single generation of memory technology which limits mix-and-match of different technologies with different cost-per-GB and bandwidth vs. latency profiles; (b) memory capacity comes at power-of-two granularity which limits finer grain memory capacity sizing; (c) there are limited bandwidth vs. capacity points per DRAM generation (Figure 5.4). This forces higher memory capacity in order to get more bandwidth on the system. Such tight coupling between CPU and memory subsystem restricts the flexibility in designing efficient memory hierarchies and leads to stranded compute, network, and/or memory resources. Prior bus interfaces that allow memory expansion are also proprietary to some extent [50, 169, 11] and not commonly supported across all the CPUs [38, 62, 22]. Besides, high latency characteristics and lack of coherency limit their viability in hyperscalers.

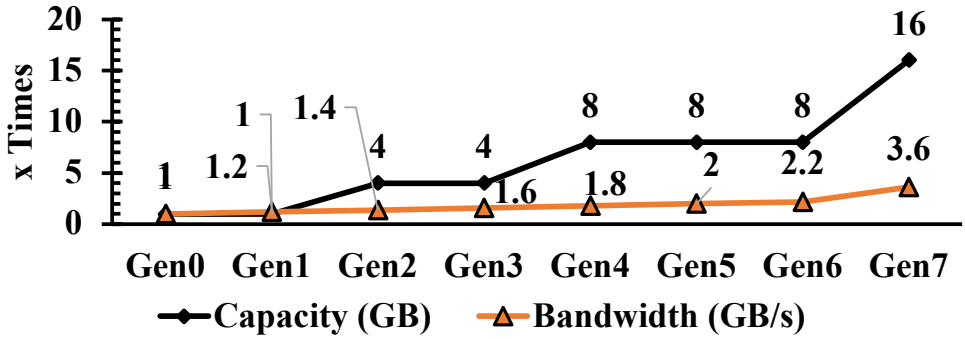
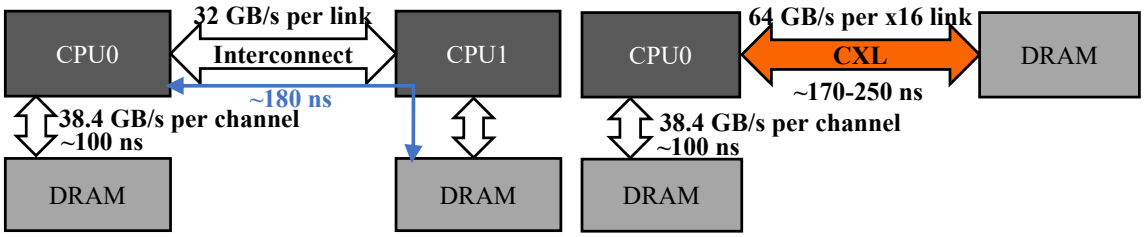


Figure 5.4: Memory bandwidth and capacity increase over time.

CXL for Designing Tiered-Memory Systems. CXL [27] is an open, industry-supported interconnect based on the PCI Express (PCIe) interface. It enables high-speed, low latency communication between the host processor and devices (e.g., accelerators, memory buffers, smart I/O devices, etc.) while expanding memory capacity and bandwidth. CXL provides byte addressable memory in the same physical address space and allows transparent memory allocation using standard memory allocation APIs. It allows cache-line granularity access to the connected devices and underlying hardware maintains coherency and consistency. With PCIe 5.0, CPU to CXL interconnect bandwidth will be similar to the cross-socket interconnects (Figure 5.5) on a dual-socket machine. CXL-Memory access latency is also similar to the NUMA access latency. CXL adds around 50-100 nanoseconds of extra latency over normal DRAM access. This NUMA-like behavior with main memory-like access semantics makes CXL-Memory a good candidate for the slow-tier in datacenter memory hierarchies.

CXL solutions are being developed and incorporated by leading chip providers [1, 12, 59, 70, 30, 69]. All the tools, drivers, and OS changes required to support CXL are open sourced so that anyone can contribute and benefit directly without relying on single supplier solutions. CXL relaxes most of the memory subsystem limitations mentioned earlier. It enables flexible memory subsystem designs with desired memory bandwidth, capacity, and cost-per-GB ratio based on workload demands. This helps scale compute and memory resources independently and ensure a better utilization of stranded resources.

Scope of CXL-Based Tiered-Memory Systems. Datacenter workloads rarely use all of the memory all the time [202, 41, 4, 292, 301]. Often an application allocates a large amount of memory but accesses it infrequently [202, 226]. We characterize four popular applications in our production server fleet and find that 55-80% of an application's allocated memory remains idle within any two minutes interval (§5.3.2). Moving this cold memory to a slower memory tier can create space for more hot memory pages to operate on the fast memory tier and improve



(a) **Without CXL** (b) **With CXL**

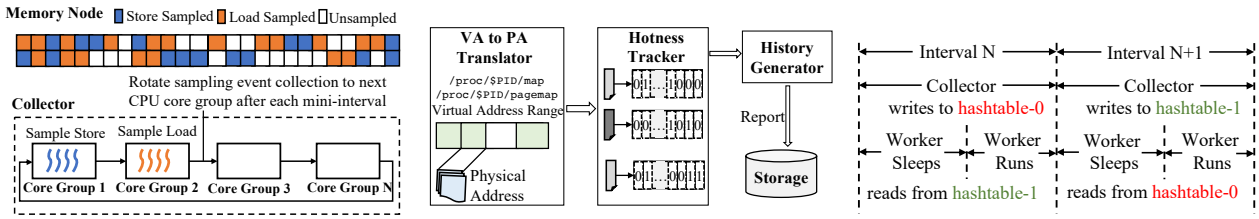
Figure 5.5: CXL-System compared to a dual-socket server.

application-level performance. Besides, it also allows reducing TCO by flexible server design with smaller fast memory tier and larger but cheaper slow memory tier.

As CXL-attached slow memory can be of any technology (e.g., DRAM, NVM, LPDRAM, etc.), for the sake of generality, we will call a memory directly attached to a CPU as local memory and a CXL-attached memory as CXL-Memory.

Lightweight Characterization of Datacenter Applications. In a hyperscaler environment, different types of rapidly evolving applications consume a production server's resources. Applications can have different requirements, e.g., some can be extremely latency sensitive, while others can be memory bandwidth sensitive. Sensitivity towards different memory page types can also vary for different applications. To understand the scope of tiered-memory system for existing datacenter applications, we need to characterize their memory usage pattern and quantify the opportunity of memory offloading at different memory tiers for different page types. This insight can help system admins decide on the flexible and optimized memory configurations to support different workloads.

Existing memory characterization tools fall short of providing these insights. For example, access bit-based mechanism [32, 46, 291] cannot track detailed memory access behavior because it tells whether a given page is accessed within a certain period of time. Even if a page gets multiple accesses within a tracking cycle, IPT-based tools will count that as a single access. Moreover, IPT only provides information in the physical address space – we cannot track memory allocation/deallocation if a physical page is re-used by multiple virtual pages. The overhead of such tools significantly increases with the application's memory footprint size. Even for application's with 10s of GB working set size, the overhead of IPT-based tool can be 20–90% [**?**]. Similarly, complex PEBS-based user-space tools [261, 123, 303, 274] lead to high CPU overheads (more than 15% per core) and often slow down the application. None of these tools generate page type-aware heat map.



(a) **Collector** (b) **Worker** (c) **Workflow within a Cycle**

Figure 5.6: Overview of Chameleon components (left) and workflow (right)

5.3 Characterizing Datacenter Applications

To understand the scope of tiered-memory in hyperscale applications, we develop Chameleon, a light-weight user-space memory access behavior characterization tool. The objective of developing Chameleon is to allow users hop on any existing datacenter production machine and readily deploy it without disrupting the running application(s) and modifying the underline kernel. Chameleon's overhead needs to be comparatively lower such that it does not notably affect a production application's behavior. Chameleon's prime use case is to understand an application's memory access behavior, i.e., what fraction of an application's memory remains hot-warm-cold, how long a page survive on a specific temperature tier, how frequently they get accessed and so on. In practice, to characterize a certain type of application, we expect to run Chameleon for a few hours on a tiny fraction of servers in the whole fleet.

Considering the above objectives, we design Chameleon with two primary components – a Collector and a Worker– running simultaneously on two different threads. Collector utilizes the PEBS mechanism of modern CPUs to collect hardware-level performance events related to memory accesses. Worker uses the sampled information to generate insights.

Collector. Collector samples last level cache (LLC) misses for demand loads (event `MEM_LOAD_RETIRED.L3_MISS`) and optionally TLB misses for demand stores (event `MEM_INST_RETIRED.STLB_MISS_STORES`). Sampled records provide us with the PID and virtual memory address for the memory access events. Like any other sampling mechanism, accuracy of PEBS depends on the sampling rate – frequent samples provide higher accuracy. High sampling rate, however, incurs higher performance overhead directly on application threads and demands more CPU resources for Chameleon's Worker thread. In our fleets, sampling rate is configured as one sample for every 200 events, which appears as a good trade-off between overhead and accuracy. Note the choice of the sampling event on the store side is due to hardware limitations, i.e., there is no precise event for LLC-missed stores, likely because stores are marked complete once TLB translation is done in modern high-end CPUs. We have conveyed the concern on this limitation to major x86 vendors in multiple occasions.

To improve flexibility, the Collector divides all CPU cores into a set of groups and enables sampling on one or more group(s) at a time (Figure 5.6a). After each `mini_interval` (by default, 5 seconds), the sampling thread rotates to the next core group. This duty-cycling helps further tune the trade-off between overhead and accuracy. It also allows sampling different events on different core groups. For example, for latency-critical applications, one can choose to sample half of the cores at a time. On the other hand, for store-heavy applications, one can enable load sampling on half of the cores and store sampling on the other half at the same time.

The Collector reads the sampling buffer and writes into one of the two hash tables. After each `interval` (by default, 1 minute), the Collector wakes up the Worker to process data in current hash table and moves to the other hash table for storing next interval's sampled data.

Worker. The Worker (Figure 5.6b) runs on a separate thread to read page access information and generate insights on memory access behavior. It considers the address of a sampled record as a virtual page access where the page size is defined by the OS. This makes it generic to systems with any page granularities (e.g., 4KB base page, 2MB huge page, etc.). To generate statistics on both virtual- and physical-spaces, the Worker finds the corresponding physical page mapped to the sampled virtual page. This address translation can cause high overhead if the target process's working set size is extremely large (e.g., terabyte-scale). One can disable the Worker's physical address translation feature and characterize an application only on its virtual-space access pattern.

For each page, a 64-bit bitmap tracks its activeness within an interval. If a page is active within an interval, the corresponding bit is set. At the end of each interval, the bitmap is left-shifted one bit to track for a new interval. One can use multiple bits for one interval to capture the page access frequency, at the cost of supporting shorter history. After generating the statistics and reporting them, the Worker sleeps (Figure 5.6c).

To characterize an application deployed on hundreds of thousands of servers, we run Chameleon on a small set of servers only for a few hours. During our analysis, we chose servers where system-wide CPU and memory usage does not go above 80%. In such production environments, we do not notice any service-level performance impact while running Chameleon. CPU overhead is within 3–5% of a single core. However, on a synthetic memory bandwidth sensitive workload that uses all CPU cores so that Chameleon needs to contend for CPU, we lose 7% performance due to profiling.

5.3.1 Production Workload Overview

We use Chameleon to characterize most popular memory-bound applications running for years across our production server fleet serving live traffic on four diverse service domains. These workloads constitute a significant portion of the server fleet and represent a wide variety of our workloads [285, 284]. **Web** implements a Virtual Machine to serve web requests. Web1 is a HipHop

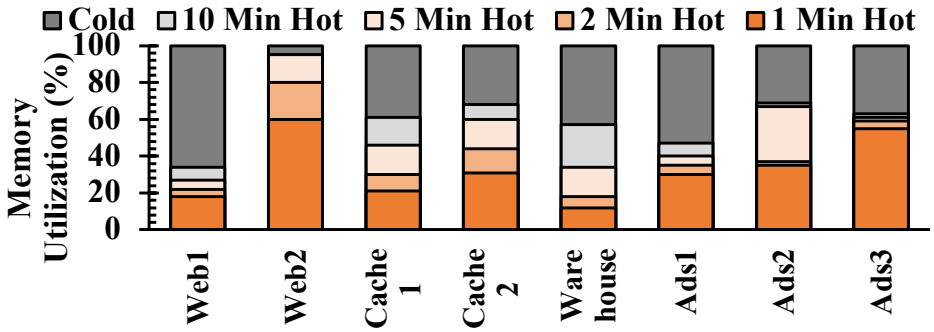


Figure 5.7: Application memory usage over last N mins.

Virtual Machine (HHVM)-based and Web2 is a Python-based service. **Cache** is a large distributed-memory object caching service lying between the web and database tiers for low-latency data-retrieval. **Data Warehouse** is a unified computing engine for parallel data processing on compute clusters. This service manages and coordinates the execution of long and complex batch queries on data across a cluster. **Ads** are compute heavy workloads that retrieve in-memory data and perform machine learning computations.

5.3.2 Page Temperature

In datacenter applications, a significant amount of allocated memory remains cold beyond a few minutes of intervals (Figure 5.7). Web, Cache, and Ads use 95–98% of the system's total memory capacity, but within a two-minute interval, they use 22–80% of the total allocated memory on average. Data Warehouse is a compute-heavy workload where a specific computation operation can span even terabytes of memory. This workload consumes almost all the available memory within a server. Even here, on average, only 20% of the accessed memory is hot within a two-minute interval.

Observation: A significant portion of a datacenter application's accessed memory remain cold for minutes. Tiered memory system can be a good fit for such cold memory if page placement mechanism can move these cold pages to a lower memory tier.

5.3.3 Temperature Across Different Page Types

Applications consume different types of pages based on application logic and execution demand. However, the fraction of anons (anonymous pages) remain hot is higher than the fraction of files (file pages). For Web, within a two-minute interval, 35–60% of the total allocated anons remain hot; for files, in contrast, it is only 3–14% of the total allocation (Figure 5.8).

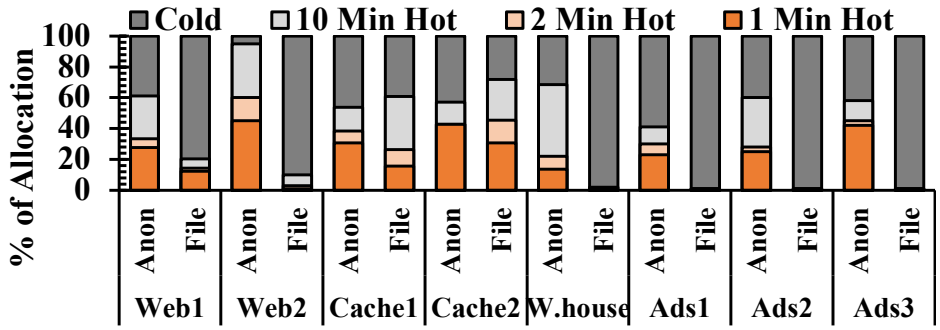
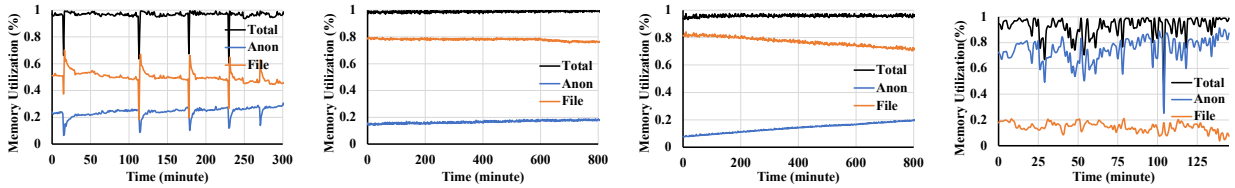


Figure 5.8: Anon pages tends to be hotter than file pages.



(a) **Web1** (b) **Cache1** (c) **Cache2** (d) **Data Warehouse**

Figure 5.9: Memory usage over time for different applications

Cache applications use `tmpfs` [77] for a fast in-memory lookup. Anons are used mostly for processing queries. As a result, file pages contribute significantly to the total hot memory. However, for Cache1, 40% of the anons get accessed within every two minutes, while the fraction for file is only 25%. For Cache2, the fraction of anon and file usage is almost equal within a two-minute time window. However, within a minute interval, even for Cache2, higher fraction of anons (43%) remain hot over file pages (30%).

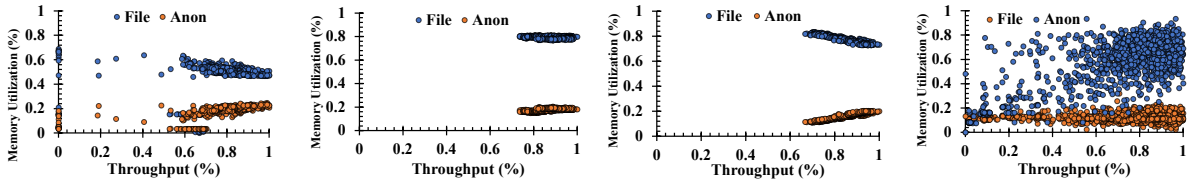
Data Warehouse and Ads use anon pages for computation. The file pages are used for writing intermediate computation data to the storage device. As expected, almost all of hot memories are anons where almost all of the files remain cold.

Observation: A large fraction of anon pages is hot, while file pages are comparatively colder within short intervals.

Due to space constraints, we focus on a subset of these applications. In our analysis, we find the similar behavior from the rest of the applications.

5.3.4 Usage of Different Page Types Over Time

When the Web service starts, it loads the VM's binary and bytecode files into memory. As a result, at the beginning, file caches occupy a significant portion of the memory. Overtime, anon usage slowly grows and file caches get discarded to make space for the anon pages (Figure 5.9a).



(a) **Web1** (b) **Cache1** (c) **Cache2** (d) **Data Warehouse**

Figure 5.10: Workloads' sensitivity towards anons and files varies. High memory capacity utilization provides high throughput.

Cache applications mostly use file caches for in-memory look-ups. As a result, file pages consume most of the allocated memory. For Cache1 and Cache2 (Figure 5.9b-5.9c), the fraction of files hovers around 70–82%. While the fraction of anon and file is almost steady, if at any point, anon usage grows, file pages are discarded to accommodate newly allocated anons.

For Data Warehouse workload, anon pages consume most of the allocated memory – 85% of the total allocated memory are anons and rest of the 15% are file pages (Figure 5.9d). The usage of anon and file pages mostly remains steady.

Observation: Although anon and file usage may vary over time, applications mostly maintain a steady usage pattern. Smart page placement mechanisms should be aware of page type when making placement decisions.

5.3.5 Impact of Page Types on Performance

Figure 5.10 shows what fractions of different page types are used to achieve a certain application-level throughput. Memory-bound application's throughput improves with high memory utilization. However, workloads have different levels of sensitivity toward different page types. For example, Web's throughput improves with the higher utilization of anon pages (Figure 5.10a).

For Cache, `tmpfs` is allocated during initialization period. Besides, Cache1 uses a fixed amount of anons throughout its life cycle. As a result, we cannot observe any noticeable relation between anon or file usage and the application throughput (Figure 5.10b). However, for Cache2, we can see high throughput is achieved with comparatively higher utilization of anons (Figure 5.10c). Similarly, Data Warehouse application maintains a fixed amount of file pages. However, it consumes different amount of anons at different execution period and the highest throughput is achieved when the total usage of anons reaches to its highest point (Figure 5.10d).

Observation: Workloads have different levels of sensitivity toward different page types that varies over time.

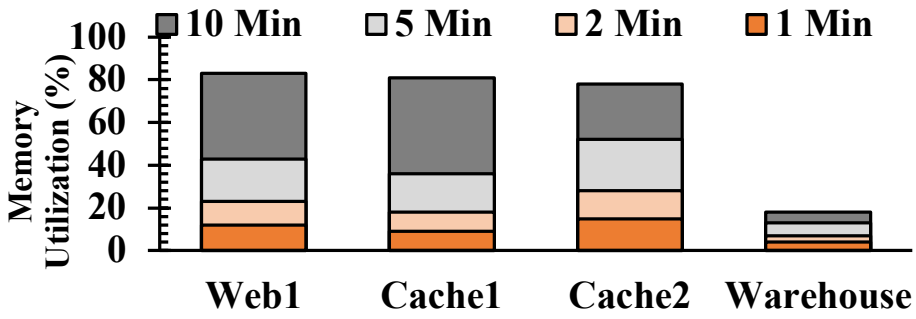


Figure 5.11: Fraction of pages re-accessed at different intervals.

5.3.6 Page Re-access Time Granularity

Cold pages often get re-accessed at a later point. Figure 5.11 shows the fraction of pages that become hot after remaining cold for a certain interval. For Web, almost 80% of the pages are re-accessed within a ten-minute interval. This indicates Web mostly repurposes pages allocated at an earlier time. Same goes for Cache – randomly offloading cold memory can impact performance as a good chunk of colder pages get re-accessed within a ten-minute window.

However, Data Warehouse shows different characteristics. For this workload, anons are mostly newly allocated – within a ten-minute interval, only 20% of the hot file pages are previously accessed. Rest of them are newly allocated.

Observation: Cold page re-access time varies for workloads. Page placement on a tiered memory system should be aware of this and actively move hot pages to lower memory nodes to avoid high memory access latency.

From above observations, tiered memory subsystems can be a good fit for datacenter applications as there exists significant amount of cold memory with steady access patterns.

5.4 Design Principles of TPP

With the advent of CXL technologies, hyperscalers are embracing CXL-enabled heterogeneous tiered-memory system where different memory tier has different performance characteristics [208, 163]. For performance optimization in such systems, transparent page placement mechanism (TPP) is needed to handle pages with varied hotness characteristics on appropriate temperature tiers. To design TPP for next-generation tiered-memory systems, we consider following questions:

- What is an ideal layer to implement TPP functionalities?
- How to detect page temperature?

- What abstraction to provide for accessing CXL-Memory?

In this section, we discuss the rationale and trade-offs behind design choices for TPP.

Implementation Layer. Application-transparent page placement mechanism can be employed both in the user- and kernel-space. In user-space, a Chameleon-like tool can be used to detect page temperatures and perform NUMA migrations using user-space APIs (e.g., `move_pages()`). To identify what to migrate, the migration tool needs to implement user-space page lists and history management. This technique entails overheads due to user-space to kernel-space context switching. It also adds processing overheads due to history management in user space. Besides, there are memory overheads due to page information management in user-space that may not scale with large working sets. While this is acceptable for profiling tools that run for short intervals on a small sample of servers in the fleet, it can be prohibitively expensive when they run continuously on all production fleet. Considering these, we design TPP as a kernel feature as we think it's less complex to implement and more performant over user-space mechanisms.

Page Temperature Detection. There are several different techniques that can potentially be used for page temperature detection including PEBS, Page Poisoning, and NUMA Balancing. PEBS can be utilized in kernel-space to detect page temperature. However, as PEBS counters are not standardized across CPU vendors, a generic precise event-based kernel implementation for page temperature detection that works across all hardware platforms is not feasible. Additionally, limited number of perf counters are supported in CPUs and are generally required to be exposed in user-space. More importantly, as mentioned earlier, even with optimizations, PEBS-based profiling is not good enough as an always-running component of TPP for high pressure workloads.

Sampling and poisoning a few pages within a memory region to track their access events is another well-established approach [88, 81, 60, 45] for finding hot/cold pages. To detect a page access, IPT-based [45, 81] approach needs to clear the page's `accessed bit` and flush the corresponding TLB entry. This requires monitoring `accessed bits` at high frequency which results in unacceptable slowdowns [88, 81]. Thermostat [88] solves this problem by sampling at 2MB page granularity which makes it effective specifically for huge-pages. One of our design goals behind TPP is that it should be agnostic to page size. In our production environment, application owners generally pre-allocate 2MB and 1GB pages and use them to allocate text regions (code), static data structures, and slab pools that serve request allocations. In most cases, these large pages are hot and should never get demoted to CXL-Memory.

NUMA Balancing (also known as AutoNUMA) [60] is transparent to OS page sizes. It generates a minor page fault when the sampled page gets accessed. Periodically incurring page faults on most frequently accessed pages can lead to high overheads. To address this, when designing TPP, we chose to only leverage minor page fault as a temperature detection mechanism for CXL-Memory. As CXL-Memory is expected to hold warm and cold pages, this will keep the overhead

of temperature detection low. For cold page detection in local memory node, we find Linux's existing LRU-based age management mechanism is lightweight and quite efficient.

Empirically, we do not see any potential benefit to leverage a more sophisticated page temperature detection mechanism. In our experiments (§5.6), using kernel LRUs for on-the-fly profiling works well. Combining LRUs and NUMA Balancing, we can detect most hot pages in CXL-Memory at virtually zero overhead as presented in Figure 5.14 and Table 5.1.

Memory Abstraction for CXL-Memory. One can use CXL-Memory as a swap-space to host colder memory using existing in-memory swapping mechanisms [76, 37, 87, 28]. TMO [301] is one such swap-based mechanism that detects cold pages in local memory and moves them to swap-space that is referred to as (z)swap pool. However, we do not plan to use CXL-Memory as an in-memory swap device. In such a case, we will effectively lose CXL's most important feature, i.e., load/store access semantics at cache-line granularity. With swap abstraction, every access to a (z)swapped page will incur a major page fault and we have to read the whole page from CXL-Memory. This will significantly increase the effective access latency far over 200ns and make CXL-Memory less attractive. We chose to build TPP so that applications can leverage load-store semantics when accessing warm or cold data from CXL-Memory.

While the swap semantics of TMO are not desirable in CXL-Memory page placement context, the memory saving through feedback-driven reclamation is still valuable. We think TMO as an orthogonal and complimentary tool to TPP as it operates one layer above TPP (§5.6.3.2). TMO runs in user-space and keeps pushing for memory reclamation, while TPP runs in kernel-space and optimizes page placement for allocated memory between local and CXL-Memory tiers.

5.5 TPP for CXL-Memory

An effective page placement mechanism should efficiently offload cold pages to slower CXL-Memory while aptly identify trapped hot pages in CXL-node and promote them to the fast memory tier. As CXL-Memory is CPU-less and independent of the CPU-attached memory, it should be flexible enough to support heterogeneous memory technologies with varied characteristics. Page allocation to a NUMA node should not frequently halt due to the slower reclamation mechanism to free up spaces. Besides, an effective policy should be aware of an application's sensitivity toward different page types.

Considering the datacenter workload characteristics and our design objectives, we propose TPP– a smart OS-managed mechanism for tiered-memory system. TPP places 'hotter' pages in local memory and moves 'colder' pages in CXL-Memory. TPP's design-space can be divided across four main areas – **(a)** lightweight demotion to CXL-Memory, **(b)** decoupled allocation and reclamation paths, **(c)** hot-page promotion to local nodes, and **(d)** page type-aware memory allocation.

5.5.1 Migration for Lightweight Reclamation

Linux tries to allocate a page to the memory node local to a CPU where the process is running. When a CPU's local memory node fills up, default reclamation pages-out to swap device. In such a case, in a NUMA system, new allocations to local node halts and takes place on CXL-node until enough pages are freed up. The slower the reclamation is, the more pages end up being allocated to the CXL-node. Besides, invoking paging events in the critical path worsens the average page access latency and impacts application performance.

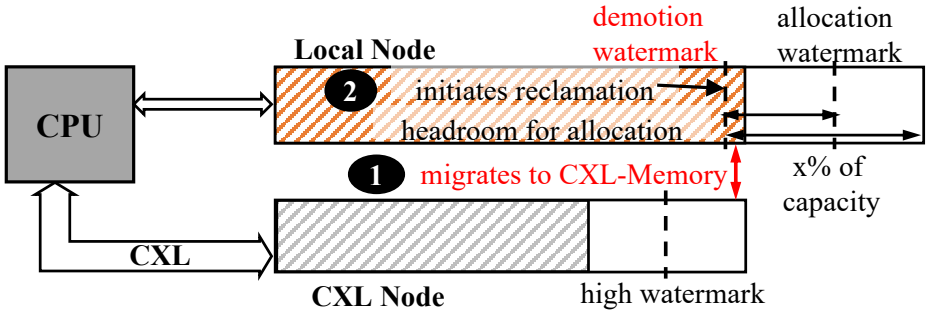


Figure 5.12: TPP decouples the allocation and reclamation logics for local memory node. It uses migration for demotion.

To enable a light-weight page reclamation for local nodes, after finding the reclamation-candidates, instead of invoking swapping mechanism, we put them in to a separate demotion list and try to migrate them to the CXL-node asynchronously (❶ in Figure 5.12). Migration to a NUMA node is orders of magnitude faster than swapping. We use Linux's default LRU-based mechanism to select demotion candidates. However, unlike swapping, as demoted pages are still available in-memory, along with inactive file pages, we scan inactive anon pages for reclamation candidate selection. As we start with the inactive pages, chances of hot pages being migrated to CXL-node during reclamation is very low unless the local node's capacity is smaller than the hot portion of working set size. If a migration during demotion fails (e.g., due to low memory on the CXL-node), we fall back to the default reclamation mechanism for that failed page. As allocation on CXL-node is not performance critical, CXL-nodes use the default reclamation mechanism (e.g., pages out to the swap device).

If there are multiple CXL-nodes, the demotion target is chosen based on the node distances from the CPU. Although other complex algorithms can be employed to dynamically choose the demotion target based on a CXL-node's state, this simple distance-based static mechanism turns out to be effective.

5.5.2 Decoupling Allocation and Reclamation

Linux maintains three watermarks (`min`, `low`, `high`) for each memory zone within a node. If the total number of free pages for a node goes below `low_watermark`, Linux considers the node is under memory pressure and initiates page reclamation for that node. In our case, TPP demotes them to CXL-node. New allocation to local node halts till the reclaimer frees up enough memory to satisfy the `high_watermark`. With high allocation rate, reclamation may fail to keep up as it is slower than allocation. Memory retrieved by the reclaimer may fill up soon to satisfy allocation requests. As a result, local memory allocations halt frequently, more pages end up in CXL-node which eventually degrades application performance.

In a multi-NUMA system with severe memory constraint, we should proactively maintain a reasonable amount of free memory headroom on the local node. This helps in two ways. First, new allocation bursts (that are often related to request processing and, therefore, both short-lived and hot) can be directly mapped to the local node. Second, local node can accept promotions of trapped hot pages on CXL-nodes.

To achieve that, we decouple the logic of 'reclamation stop' and 'new allocation happen' mechanism. We continue the asynchronous background reclamation process on local node until its total number of free pages reaches `demotion_watermark`, while new allocation can happen if the free page count satisfies a different watermark – `allocation_watermark` (❷ in Figure 5.12). Note that demotion watermark is always set to a higher value above the allocation and low watermark so that we always reclaim more to maintain the free memory headroom.

How aggressively one needs to reclaim often depends on the application behavior and available resources. For example, if an application has high page allocation demand, but a large fraction of its memory is infrequently accessed, aggressive reclamation can help maintain free memory headroom. On the other hand, if the amount of frequently accessed pages is larger than the local node's capacity, aggressive reclamation will thrash hot memory across NUMA nodes. Considering these, to tune the aggressiveness of the reclamation process on local nodes, we provide a user-space `sysctl` interface (`/proc/sys/vm/demote_scale_factor`) to control the free memory threshold for triggering the reclamation on local nodes. By default, its value is empirically set to 2% that means reclamation starts as soon as only a 2% of the local node's capacity is available to consume. One can use workload monitoring tools [301] to dynamically adjust it.

5.5.3 Page Promotion from CXL-Node

Due to increased memory pressure on local nodes, new pages may often get allocated to CXL-nodes. Besides, demoted pages may also become hot later as discussed in §5.3.6. Without any promotion mechanism, hot pages will always be trapped in CXL-nodes and hurt application per-

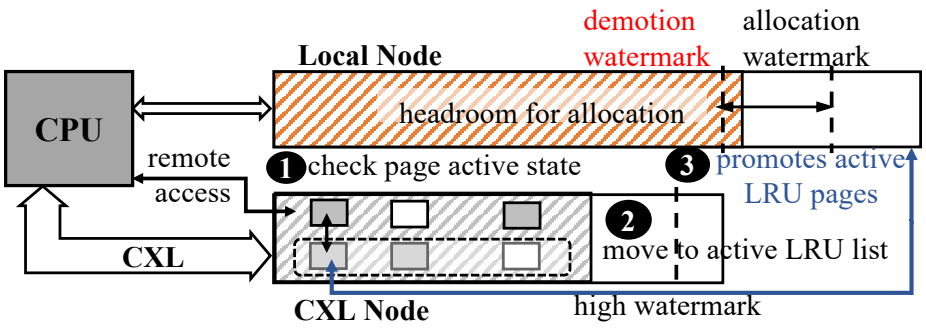


Figure 5.13: TPP promotes a page considering its activity state.

formance. To promote such pages, we augment Linux's NUMA Balancing [60].

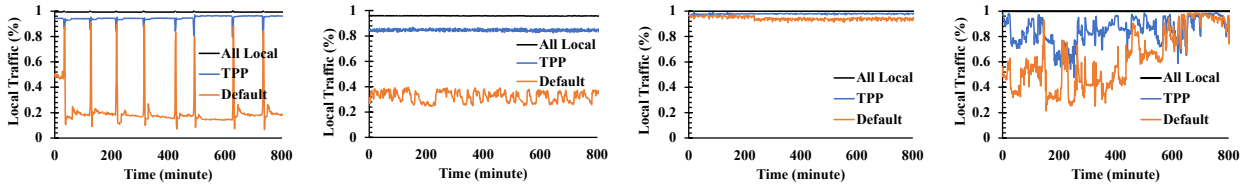
NUMA Balancing for CXL-Memory. In NUMA Balancing, a kernel task routinely samples a subset of a process's memory (by default, 256MB of pages) on each memory node. When a CPU accesses a sampled page, a minor page-fault is generated (known as NUMA hint fault). Pages that are accessed from a remote CPU are migrated to that CPU's local memory node (known as promotion). In a CXL-System, it is not reasonable to promote a local node's hot memory to other local or CXL-nodes. Besides, as sampling pages to find a local node's hot memory generates unnecessary NUMA hint fault overheads, we limit sampling only to CXL-nodes.

While promoting a CXL-node's trapped hot pages, we ignore the allocation watermark checking for the target local node. This creates more memory pressure to initiate the reclamation of comparatively colder pages on that node. If a system has multiple local nodes, we select the node where the task is running. When applications share multiple memory nodes, we choose local node with the lowest memory pressure.

Ping-Pong due to Opportunistic Promotion. When a NUMA hint fault happens on a page, default NUMA balancing instantly promotes the page without checking its active state. As a result, pages with very infrequent accesses can still be promoted to the local node. Once promoted, these type of pages may shortly become the demotion candidate if the local nodes are always under pressure. Thus, promotion traffic generated from infrequently accessed pages can easily fill up the local node's free space and generate a higher demotion traffic for CXL-nodes. This unnecessary traffic can negatively impact an application's performance.

Apt Identification of Trapped Hot Pages. To solve this ping-pong issue, instead of instant promotion, we check a page's age through its position in the LRU list maintained by the OS. If the faulted page is in inactive LRU, we do not consider the page for promotion instantly as it might be an infrequently accessed page. We consider the faulted page as a promotion candidate only if it is found in the active LRUs (❶ in Figure 5.13). This significantly reduces the promotion traffic.

However, OS uses LRU lists for reclamation. If a memory node is not under pressure and



(a) **Web1** (b) **Cache1** (c) **Cache2** (d) **Data Warehouse**

Figure 5.14: Fast cold page demotion and effective hot page promotion allow TPP to serve most of the traffics from the local node.

reclamation does not kick in, then pages in inactive LRU list do not automatically move to the active LRU list. As CXL-nodes may not always be under pressure, faulted pages may often be found in the inactive LRU list and, therefore, bypass the promotion filter. To address this, whenever we find a faulted page on the inactive LRU list, we mark the page as accessed and move it to the active LRU list immediately (❷ in Figure 5.13). If the page still remains hot during the next NUMA hint fault, it will be in the active LRU, and promoted to the local node (❸ in Figure 5.13).

This helps TPP add some hysteresis to page promotion. Besides, as Linux maintains separate LRU lists for anon and file pages, different page types have different promotion rates based on their respective LRU sizes and activeness. This speeds up the convergence of hot pages across memory nodes.

5.5.4 Page Type-Aware Allocation

The page placement mechanism we described above is generic to all page types. However, some applications can further benefit from page type-aware allocation policy. For example, production applications often perform lots of file I/O during the warm up phase and generate file caches that are accessed infrequently. As a result, cold file caches eventually end up on CXL-nodes. Not only that, local memory node being occupied by the inactive file caches often forces anons to be allocated on the CXL-nodes that may need to be promoted back later.

To resolve these unnecessary page migrations, we allow an application allocating caches (e.g., file cache, tmpfs, etc.) to the CXL-nodes preferrably, while preserving the allocation policy for anon pages. When this allocation policy is enabled, page caches generated at any point of an application's life cycle will be initially allocated to the CXL-node. If a page cache becomes hot enough to be selected as a promotion candidate, it will be eventually promoted to the local node. This policy helps applications with infrequent cache accesses run on a system with a small amount of local memory and large but cheap CXL-Memory while maintaining the performance.

5.5.5 Observability into TPP to Assess Performance

Promotion- and demotion-related statistics can help better understand the effectiveness of the page placement mechanism and debug issues in production environments. To this end, we introduce multiple counters to track different demotion and promotion related events and make them available in the user-space through `/proc/vmstat` interface.

To characterize the demotion mechanism, we introduce counters to track the distribution of successfully demoted anon and file pages. To understand the promotion behavior, we add new counters to collect information on the numbers of sampled pages, the number of promotion attempts, and the number of successfully promoted pages for each of the memory types.

To track the ping-pong issue mentioned in §5.5.3, we utilize the unused `0x40` bit in the page flag to introduce `PG_demoted` flag for demoted pages. Whenever a page is demoted its `PG_demoted` bit is set and gets cleared upon promotion. We also count the number of demoted pages that become promotion candidates. High value of this counter means TPP is thrashing across NUMA nodes. We add counters for each of the promotion failure scenario (e.g., local node having low memory, abnormal page references, system-wide low memory, etc.) to reason about where and how promotion fails.

5.6 Evaluation

We integrate TPP on Linux kernel v5.12. We evaluate TPP with a subset of representable production applications mentioned in §5.3.1 serving live traffic on tiered memory systems across our server fleet. We explore the following questions:

- How effective TPP is in distributing pages across memory tiers and improving application performance? (§5.6.1)
- What are the impacts of TPP components? (§5.6.2)
- How it performs against state-of-the-art solutions? (§5.6.3)

For each experiment, we use application-level throughput as the metric for performance. In addition, we use the fraction of memory accesses served from the local node as the key low-level supporting metric. We compare TPP against default Linux (§5.6.1), default NUMA Balancing [60], AutoTiering [189], and TMO [301] (§5.6.3) (Table 5.1). None of our experiments involve swapping to disks, the whole system has enough memory to support the workload. We use the default *local-node first, then CXL-node* allocation policy for Linux.

Workload/Throughput (%) (normalized to Baseline)	Default Linux	TPP	NUMA Balancing	AutoTiering
Web1 (2:1)	83.5	**99.5**	82.8	87.0
Cache1 (2:1)	97.0	**99.9**	93.7	92.5
Cache1 (1:4)	86.0	**99.5**	90.0	Fails
Cache2 (2:1)	98.0	**99.6**	94.2	94.6
Cache2 (1:4)	82.0	**95.0**	78.0	Fails
Data Warehouse (2:1)	99.3	**99.5**	–	–

Table 5.1: TPP is effective over its counterparts. It reduces memory access latency and improves application throughput.

Experimental Setup. We deploy a number of pre-production x86 CPUs with FPGA-based CXL-Memory expansion card that support CXL 1.1. Memory attached to the expansion card shows up to the OS as a CPU-less NUMA node. Although our current FPGA-based CXL cards have around 250ns higher latency than our eventual target, we use them for the functional validation.

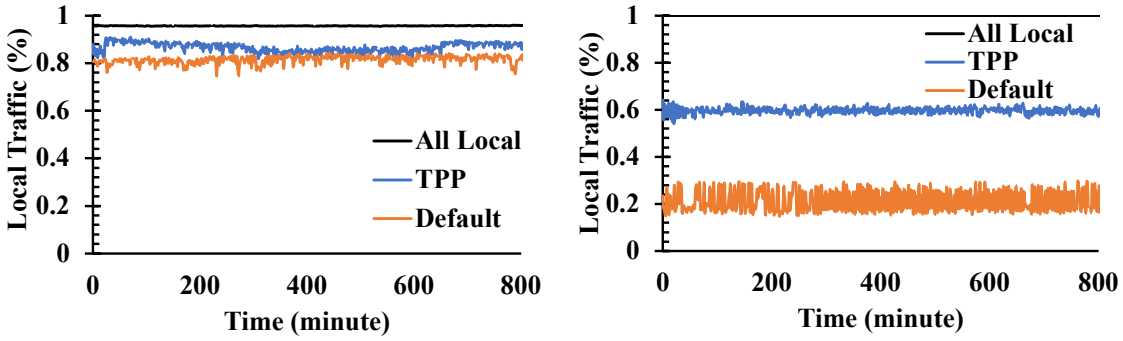
We have confirmation from two major x86 CPU vendors that the access latency to CXL-Memory is similar to the remote latency on a dual-socket system. For performance evaluation, we primarily use dual-socket systems and configure them to mimic our target CXL-enabled system's characteristics (one memory node with all active CPU cores and one CPU-less memory node) according to the guidance of our CPU vendors. For baseline, we disable the memory node and CPU cores on a socket while enabling sufficient memory on another socket. Here, single memory node serves the whole working set.

We use two memory configurations where the ratio of local node and CXL-Memory capacity is **(a)** 2:1 (§5.6.1.1) and **(b)** 1:4 (§5.6.1.2). Configuration **(a)** is similar to our current CXL-enabled production systems where local node is supposed to serve all the hot working set. We use configuration **(b)** to stress-test TPP on a constrained memory scenario – only a fraction of the total hot working set can fit on the local node and hot pages are forced to move to the CXL-node.

5.6.1 Effectiveness of TPP

5.6.1.1 Default Production Environment (2:1 Configuration).

Web. Web1 performs lots of file I/O during initialization and fills up the local node. Default Linux is 44× slower than TPP during freeing up the local node. As a result, new allocation to the local node halts and anons get allocated to the CXL-node and stay there forever. In default Linux, on average, only 22% of total memory accesses are served from the local node (Figure 5.14a). As a result, throughput drops by 16.5%.



(a) **Cache1** (b) **Cache2**

Figure 5.15: Effectiveness of TPP under memory constraint.

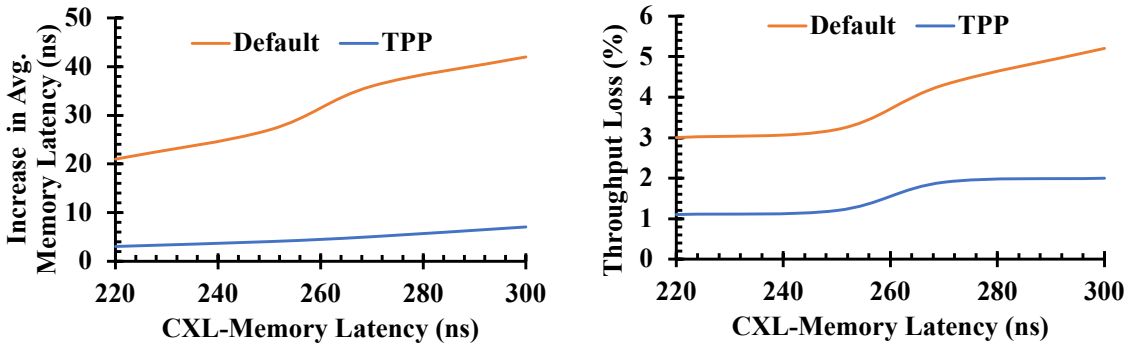
Active and faster demotion helps TPP move colder file pages to CXL-node and allow more anon pages to be allocated in the local node. Here, 92% of the total anon pages are served from the local node. As a result, local node serves 90% of total memory accesses. Throughput drop is only 0.5%.

Cache. Cache applications maintain a steady ratio of anon and file pages throughout their life cycle. Almost all the anon pages get allocated to the local node from the beginning and served from there. For Cache1, with default Linux, only 8% of the total hot pages are trapped in CXL-node. As a result, application performance remains very close to the baseline – performance regression is only 3%. TPP can even minimize this performance gap (throughput is 99.9% of the baseline) by promoting all the trapped hot files and improving the fraction of traffic served from the local node (Figure 5.14b).

Although most of the Cache2's anon pages reside in the local node on a default Linux kernel, all of them are not always hot – only 75% of the total anon pages remain hot within a two-minute interval. TPP can efficiently detect the cold anon pages and demote them to the CXL-node. This allows the promotion of more hot file pages. On default Linux, local node serves 78% of the memory accesses (Figure 5.14c) and cause 2% throughput regression. TPP improves the fraction of local traffic to 91%. Throughput regression is only 0.4%.

Data Warehouse. This workload generates file caches to store the intermediate processing data. File caches mostly remain cold. Besides, only one third of the total anon pages remain hot. Our default production configuration is good enough to serve all the hot memory from the local node. Default Linux and TPP perform the same (0.5–0.7% throughput drop).

TPP improves the fraction of local traffic by moving relatively hotter anon pages to the local node. With TPP, 94% of the total anon pages reside on the local node, while the default kernel hosts only 67%. To make space for the hot anon pages, cold file pages are demoted to the CXL-node.



(a) **Avg. Memory Access Latency** (b) **Impact on Throughput**

Figure 5.16: TPP benefits CXL-node with varied latency traits.

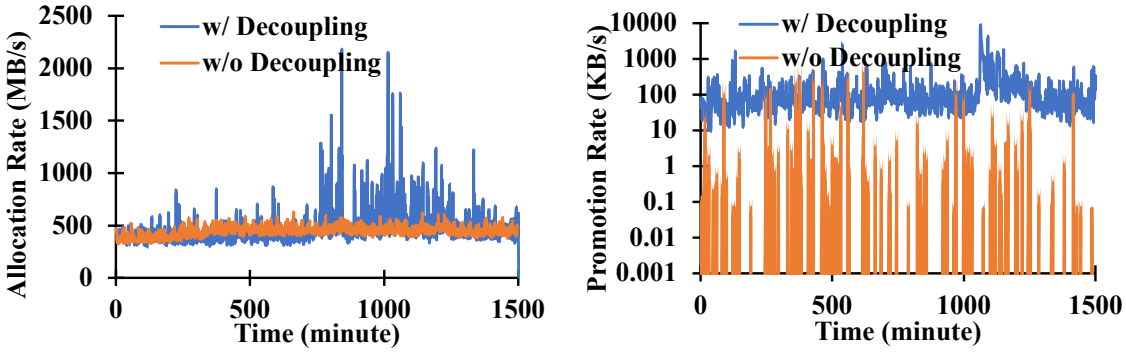
TPP allows 4% higher local node memory accesses over Linux (Figure 5.14d).

5.6.1.2 Large Memory Expansion with CXL (1:4 Configuration).

Extreme setups like 1:4 configuration allow flexible server design with DRAM as a small-sized local node and CXL-Memory as a large but cheaper memory. As in our production, such a configuration is impractical for Web and Data Warehouse, we limit our discussion to the Cache applications. Note that TPP is effective even for Web and Data Warehouse in such a setup and performs very close to the baseline.

Cache1. In a 1:4 configuration, on a default Linux kernel, file pages consume almost all the local node's capacity. 85% of the total anon pages get trapped to the CXL-node and throughput drops by 14%. Because of the apt promotion, TPP can move almost all the CXL-node's hot anon pages (97% of the total hot anon pages within a minute) to the local node. This forces less latency-sensitive file pages to be demoted to the CXL-node. Eventually, TPP stabilizes the traffic between the two nodes and local node serves 85% of the total memory accesses. This helps Cache1 achieve the baseline performance – even though the local node's capacity is only 20% of the working set size, throughput regression is only 0.5% (Figure 5.15a).

Cache2. Similar to Cache1, on default Linux, Cache2 experiences 18% throughput loss. Only 14% of the anon pages remain on the local node (Figure 5.15b). TPP can bring back almost all the remote hot anon pages (80% of the total anon pages) to local node while demoting the cold ones to the CXL-node. As Cache2 accesses lots of caches, and caches are now mostly in CXL-node, 41% of the memory traffic comes from the CXL-node. Yet, throughput drop is only 5% with TPP.



(a) **New Allocation** (b) **Promotion to Toptier Node**

Figure 5.17: Impact of decoupling allocation and reclamation.

5.6.1.3 TPP with Varied CXL-Memory Latencies.

The variation in CXL-Memory's access latency does not impact TPP much. We run Cache2 with 2:1 configuration where CXL-Memory has different latency characteristics (Figure 5.16). In all cases, due to TPP's better hot page identification and effective promotion, only a small portion (4–5%) of hot pages are served from the CXL-node. On the other hand, for default Linux, 22–25% hot pages remain trapped in the CXL-node. This increases the average memory access latency by $7\times$ for default Linux (Figure 5.16a). This, eventually, impact the application-level performance, default Linux experiences $2.2 - 2.8\times$ higher throughput loss over TPP (Figure 5.16b).

5.6.2 Impact of TPP Components

In this section, we discuss the contribution of TPP components. As a case study, we use Cache1 with 1:4 configuration.

Allocation and Reclamation Decoupling. Without this feature, reclamation on local node triggers at a later phase. With high memory pressure and delayed reclamation on local node, the benefit of TPP disappears as it fails to promote CXL-node pages. Besides, newly allocated pages are often short-lived (less than a minute life-time) and de-allocated even before being selected as a promotion candidate. Trapped CXL-node hot pages worsen the performance.

Without the decoupling feature, allocation maintains a steady rate that is controlled by the rate of reclamation (Figure 5.17a). As a result, any burst in allocations puts pages to the CXL-node. When allocation and reclamation is decoupled, TPP allows more pages on the local node – allocation rate to local node increases by $1.6\times$ at the 95^{th} percentile.

As local node is always under memory pressure, new allocations consume freed up pages instantly and promotion fails as the target node becomes low on memory after serving allocation

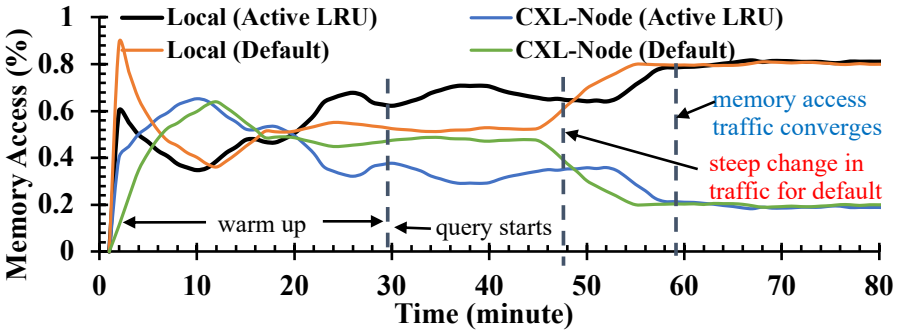
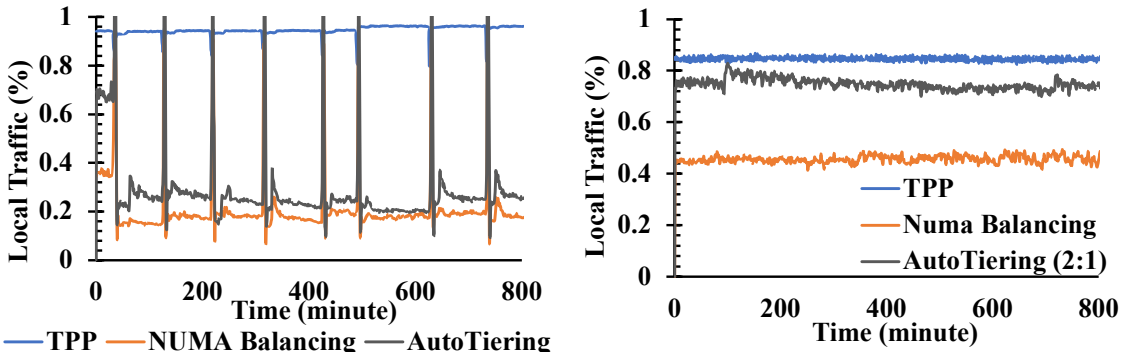


Figure 5.18: Restricting the promotion candidate based on their age reduces unnecessary promotion traffic.

Application	Configuration	% of Memory Access Traffic		Throughput w.r.t Baseline
		Local Node	CXL-node	
Web1	2:1	97%	3%	99.5%
Cache1	1:4	85%	15%	99.8%
Cache2	1:4	72%	28%	98.5%

Table 5.2: Page-type aware allocation helps applications.



(a) **Web1 on 2:1 Configuration** (b) **Cache1 on 1:4 Configuration**

Figure 5.19: TPP outperforms existing page placement mechanism. AutoTiering can't run on 1:4 configuration For Cache1, TPP on 1:4 configuration performs better than AutoTiereing on 2:1.

requests. For this reason, without the decoupling feature, promotion almost halts most of the time (Figure 5.17b). Trapped pages on the CXL-node generates 55% of the memory traffic which leads to a 12% throughput drop. With the decoupling feature, promotion maintains a steady rate of 50KB/s on average. During the surge in remote memory usage, the promotion goes as high as 1.2MB/s in the 99^{th} percentile. This helps TPP to maintain the throughput of baseline.

Active LRU-Based Hot Page Detection. Considering active LRU pages as promotion candidates helps TPP add hysteresis to page promotion. This reduces the page promotion rate by

Web1 on 2:1 Configuration	TPP-only	TPP with TMO
Migration Failure Rate (pages/sec)	20	5
CXL-node's Memory Traffic (%)	3.1%	2.7%

Table 5.3: TMO enhances TPP's memory reclamation process and improves page migration by generating more free space.

Web1 on 2:1 Configuration	TMO-only	TMO with TPP
Process Stall (Normalized to Threshold)	70%	40%
Memory Saving (% of Total Capacity)	13.5%	16.5%

Table 5.4: TPP improves TMO by effectively turning the swap action into a two-stage demote-then-swap process.

11×. The number of demoted pages that subsequently get promoted is also reduced by 50%. Although the demotion rate drops by 4%, as we are not allowing unnecessary promotion traffics to waste local memory node, now there are more effective free pages in local node. As a result, promotion success rate improves by 48%. Thus, reduced but successful promotions provide enough free spaces in local node for new allocations and improve the local node's memory access by 4%. Throughput also improves by 2.4%. The time requires to converge the traffic across memory tiers is almost similar – to reach the peak traffic on local node, TPP with active LRU-based promotion takes extra five minutes (Figure 5.18).

Cache Allocation to Remote Node Policy. For Web and Cache, preferring the file cache allocation to CXL-node can provide all-local performances even with a small-sized local node (Table 5.2). TPP is efficient enough to keep most of the effective hot pages on the local node. Throughput drop over the baseline is only 0.2–2.5%.

5.6.3 Comparison Against Existing Solutions

We compare TPP against Linux's default NUMA Balancing, AutoTiering, and TMO. We use Web1 and Cache1, two representative workloads of two different service domains, and evaluate them on target production setup (2:1 configuration) and memory-expansion setup (1:4 configuration), respectively. We omit Data Warehouse as it does not show any significant performance drop even with default Linux (§5.6.1).

5.6.3.1 TPP against NUMA Balancing and AutoTiering.

Web1. NUMA Balancing helps when the reclaim mechanism can provide with enough free pages for promotion. When the local node is low on memory, NUMA Balancing stops promoting pages

and performs even worse than default Linux because of its extra scanning and failed promotion tries. Due to $42\times$ slower reclamation rate than TPP, NUMA Balancing experiences $11\times$ slower promotion rate. Local node can serve only 20% of the memory traffic (Figure 5.19a). As a result, throughput drops by 17.2%. Due to the unnecessary sampling, its system-wide CPU overhead is 2% higher that TPP.

AutoTiering has a faster reclamation mechanism – it migrates pages with low access frequencies to CXL-node. However, with a tightly-coupled allocation-reclamation path, it maintains a fixed-size buffer to support promotion under pressure. This reserved buffer eventually fills up during a surge in CXL-node page accesses. At that point, AutoTiering also fails to promote pages and end up serving 70% of the traffic from the CXL-node. Throughput drops by 13% from the baseline.

Note that TPP experiences only a 0.5% throughput drop.

Cache1. In 1:4 configuration, with more memory pressure on local node, NUMA Balancing effectively stops promoting pages. Only 46% of the traffics are accessed from the local node (Figure 5.19b). Throughput drops by 10%.

We can not setup AutoTiering with 1:4 configuration. It frequently crashes right after the warm up phase, when query fires. We run Cache1 with AutoTiering on 2:1 configuration. TPP out performs AutoTiering even with a 46% smaller local node – TPP can serve 10% more traffic from local node and provides 7% better throughput over AutoTiering.

5.6.3.2 Comparison between TPP and TMO

TPP vs. TMO. TMO monitors application stalls during execution time because of insufficient system resources (CPU, memory, and IO). Based on the pressure stall information (PSI), it decides on the amount of memory that needs to be offloaded to the swap space. As mentioned in §5.4, using TMO for CXL-Memory's (z)swap-based abstraction is beyond our design goal. For the sake of argument, if we configure CXL-Memory as a swap-space for TMO, it will be only populated during reclamation. New page allocation can never happen there. Besides, without any fast promotion mechanism, aggressive reclamation can hurt application's performance; especially, when reclaimed pages are re-accessed through costly swap-ins. As a result, TMO throttles and can not populate most of the CXL-Memory capacity. For Web1, Cache1, and Data Warehouse in 2:1 configuration, TMO can only consume 45%, 61%, and 7% of the CXL-Memory capacity, respectively. On the other hand, TPP can use CXL-Memory for both allocation and reclamation purposes. For same applications, TPP's CXL-Memory usage is 83%, 92%, and 87%, respectively.

TPP with TMO. We run TMO with TPP and observe they are orthogonal and augment each other's behavior for Web1 on 2:1 configuration. TPP keeps most hot pages in local node; it gets slightly better when TMO is enabled (Table 5.3). TMO creates more free memory space in the system by swapping out cold pages both from local and CXL-Memory nodes. The presence of

some memory headroom in the system makes it easier for TPP to move pages around and leads to fewer page migration failures. As TPP-driven migration fails less frequently, page placement is more optimized, resulting in even fewer accesses to the CXL-node.

TMO is still able to save memory without noticeable performance overhead, as shown in Table 5.4. This is because TPP makes (z)swap a two-stage process – TMO-driven reclamation in local node will first demote victim pages to CXL-Memory before getting (z)swapped out eventually. This improves victim page selection process – semi-hot pages now get a second chance for staying in local memory when drifting down CXL-node's LRU list. As a result, TPP reduces the amount of process stall in TMO originated from major page faults (i.e., memory and IO pressure in [301]) by 30%. As TMO throttles itself based on the process stall metric, this improves memory saving by 3% (2GB in absolute terms).

5.7 Discussion and Future Research

TPP makes us production-ready to onboard our first generation of CXL-enabled tiered-memory system. We, however, foresee research opportunities with technology evolution.

Tiered Memory for Multi-tenant Clouds. In a typical cloud, when multiple tenants co-exist on a single host machine, TPP can effectively enable them to competitively share different memory tiers. When local memory size dominates the total memory capacity of the system, this may not cause much problem. However, if applications with different priorities have different QoS requirements, TPP may provide sub-optimal performance. Integrating a well-designed QoS-aware memory management mechanism over TPP can address this problem.

Allocation Policy for Memory Bandwidth Expansion. For memory bandwidth-bound applications, CPU to DRAM memory bandwidth often becomes the bottleneck. CXL's additional memory bandwidth can help by spreading memory across the top-tier and remote node. Instead of only placing cold pages into CXL-Memory, which draw very low bandwidth consumption, the optimal solution should place the right amount of bandwidth-heavy, latency-insensitive pages to CXL-Memory. The methodology to identify the ideal fraction of such working sets may even require hardware support. We want to explore transparent memory management for memory bandwidth-expansion use case in our future work.

Hardware Support for Effective Page Placement. Hardware features can further enhance performance of TPP. A memory-side cache and its associated prefetcher on the CXL ASIC might help reduce the effective latency of CXL-Memory. Hardware support for data movement between memory tiers can help reduce page migration overheads. While in our environment we do not see a high migration overheads, others may chose to put provision systems more aggressively with very small amount of local memory and high amount of CXL-Memory. For our use cases, in steady

state, the migration bandwidth is 4–16 MB/s (1–4K pages/second) which is far lower than CXL link bandwidth and also unlikely to cause any meaningful CPU overhead due to page movement.

5.8 Related Work

Tiered Memory System. With the emergence of low-latency non-DDR technologies, heterogeneous memory systems are becoming popular. There have been significant efforts in using NVM to extend main memory [144, 185, 21, 88, 249, 261, 183, 143, 230]. CXL enables an intermediate memory tier with DRAM-like low-latency in the hierarchy and brings a paradigm shift in flexible and performant server design. Industry leaders are embracing CXL-enabled tiered memory system in their next-generation datacenters [1, 12, 59, 70, 31, 30, 69].

Page Placement for Tiered Memory. Prior work explored hardware-assisted [257, 229, 234] and application-guided [299, 58, 144, 70] page placement for tiered memory systems, which may not often scale to datacenter use cases as they require hardware support or application redesign from the ground up.

Application-transparent page placement approaches often profile an application's physical [141, 183, 88, 202, 329, 81] or virtual address-space [209, 318, 304, 261] to detect page temperature. This causes high performance-overhead because of frequent invocation of TLB invalidations or interrupts. We find existing in-kernel LRU-based page temperature detection is good enough for CXL-Memory. Prior study also explored machine learning directed decisions [141, 202], user-space APIs [261, 209], and swapping [88, 202] to move pages across the hierarchy, which are either resource or latency intensive.

In-memory swapping [76, 37, 87, 28] can be used to swap-out cold pages to CXL-node. In such cases, CXL-node access requires page-fault and swapped-out pages are immediately brought back to main memory when accessed. This makes in-memory swapping ineffective for workloads that access pages at varied frequencies. When CXL-Memory is a part of the main memory, less frequently accessed pages can be on CXL-node without any page-fault overhead upon access.

Solutions considering NVM to be the slow memory tier [306, 189, 78, 167] are conceptually close to our work. Nimble [306] is optimized for huge page migrations. During migration, it employs page exchange between memory tiers. This worsens the performance as a demotion needs to wait for a promotion in the critical path. Similar to TPP, AutoTiereing [189] and work from Huang et al. [78] use background migration for demotion and optimized NUMA balancing [60] for promotion. However, their timer-based hot page detection causes computation overhead and is often inefficient, especially when pages are infrequently accessed. Besides, none of them consider decoupling allocation and reclamation paths. Our evaluation shows, this is critical for memory-bound applications to maintain their performance under memory pressure.

Disaggregated Memory. Memory disaggregation exposes capacity available in remote hosts as a pool of memory shared among many machines. Most recent memory disaggregation efforts [92, 226, 165, 224, 207, 111, 275, 267, 297, 99, 158] are specifically designed for RDMA over InfiniBand or Ethernet networks where latency characteristics are orders-of-magnitude higher than CXL-Memory. Memory managements of these systems are orthogonal to TPP– one can use both CXL- and network-enabled memory tiers and apply TPP and memory disaggregation solutions to manage memory on the respective tiers.

5.9 Conclusion

We analyze datacenter applications' memory usage behavior using Chameleon, a lightweight and robust user-space working set characterization tool, to find the scope of CXL-enabled tiered-memory system. To realize such a system, we design TPP, an OS-level transparent page placement mechanism that works without any prior knowledge on applications' memory access behavior. We evaluate TPP using diverse production workloads and find TPP improves application's performance on default Linux by 18%. TPP also outperforms NUMA Balancing and AutoTiering, two state-of-the-art tiered-memory management mechanisms, by 5–17%.

CHAPTER 6

Conclusions

Although networking technologies like InfiniBand and Ethernet continue to improve, their latency remain considerably high for providing a cache-coherent memory address space across disaggregated memory devices. CXL [27] is a new processor-to-peripheral/accelerator cache-coherent interconnect protocol that builds on and extends the existing PCIe protocol by allowing coherent communication between the connected devices.[1] It provides byte-addressable memory in the same physical address space and allows transparent memory allocation using standard memory allocation APIs. It also allows cache-line granularity access to the connected devices and underlying hardware maintains cache-coherency and consistency.

With the emergence of new hardware technologies comes the opportunity to rethink and revisit past design decisions, and CXL is no different. Earlier software solutions for memory disaggregation over RDMA are not optimized enough in CXL-based because of its much lower latency bound, especially for intra-server CXL (CXL 1.0/1.1) with 100s of nanoseconds latency. Recent works in leveraging CXL 1.0/1.1 within a server have focused on (tiered) memory pooling [225, 208] because a significant portion of datacenter application working sets can be offloaded to a slower-tier memory without hampering performance [202, 225, 226]. With the widespread usage of generation of CXL devices, we envision a huge scope for memory disaggregation.

CXL Roadmap. Today, CXL-enabled CPUs and memory devices support CXL 1.0/1.1 (Figure 6.1) that enables a point-to-point link between CPUs and accelerator memory or between CPUs and memory extenders. CXL 2.0 spec enables one-hop switching that allows multiple accelerators without (*Type-1 device*) or with memory (*Type-2 device*) to be configured to a single host and have their caches be coherent to the CPUs. It also allows memory pooling across multiple hosts using memory expanding devices (*Type-3 device*). A CXL switch has a fabric manager (it can be on-board or external) that is in charge of the device address-space management. Devices can be

[1] Prior industry standards in this space such as CCIX [22], OpenCAPI [62], Gen-Z [38] etc. have all come together under the banner of CXL consortium. While there are some related research proposals (e.g., [206]), CXL is the de facto industry standard at the time of writing this paper.

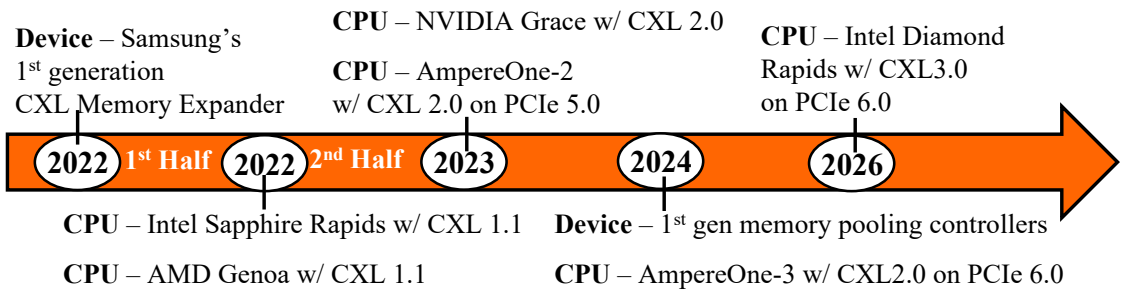


Figure 6.1: CXL roadmap paves the way for memory pooling and disaggregation in next-generation datacenter design.

hot-plugged to the switch. A virtual CXL switch partitions the CXL-Memory and isolate the resources between multiple hosts. It provides telemetry for load on each connected devices for load balancing and QoS management.

CXL 3.0 adds multi-hop hierarchical switching – one can have any complex types of network through cascading and fan-out. This expands the number of connected devices and the complexity of the fabric to include non-tree topologies, like Spine/Leaf, mesh- and ring-based architectures. CXL 3.0 supports PCIe 6.0 (64 GT/s i.e., up to 256 GB/s of throughput for a x16 duplex link) and expand the horizon of very complex and composable rack-scale server design with varied memory technologies (Figure 6.2). A new Port-Based Routing (PBR) feature provides a scalable addressing mechanism that supports up to 4,096 nodes. Each node can be any of the existing three types of devices or the new Global Fabric Attached Memory (GFAM) device that supports different types of memory (i.e., Persistent Memory, Flash, DRAM, other future memory types, etc.) together in a single device. Besides memory pooling, CXL 3.0 enables memory sharing across multiple hosts on multiple end devices. Connected devices (i.e., accelerators, memory expanders, NICs, etc.) can do peer-to-peer communicate bypassing the host CPUs.

In essence, CXL 3.0 enables large networks of memory devices. This will proliferate software-hardware co-designed memory disaggregation solutions that not only simplify and better implement previous-generation memory disaggregation solutions but also open up new possibilities.

6.1 Disaggregated Memory at Rack-Scale and Beyond: Open Challenges

Although higher than intra-server CXL latency, rack-scale CXL systems with a CXL switch (CXL 2.0) will experience much lower latency than RDMA-based memory disaggregation. With a handful of hops in CXL 3.0 setups, latency will eventually reach a couple microseconds similar to that

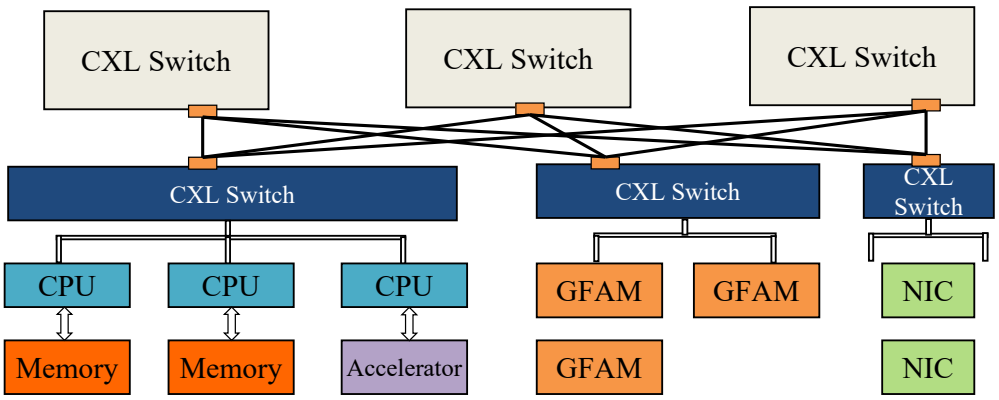


Figure 6.2: CXL 3.0 enables a rack-scale server design with complex networking and composable memory hierarchy.

found in today's RDMA-based disaggregated memory systems. For next-generation memory disaggregation systems that operate between these two extremes, i.e., rack-scale and a little beyond, many open challenges exist. We may even have to revisit some of our past design decisions (§1.1). Here we present a non-exhaustive list of challenges informed by our experience.

6.1.1 Abstractions

Memory Access Granularity. CXL enables cache-line granular memory access over the connected devices, whereas existing OS VMM modules are designed for page-granular (usually, 4KB or higher) memory access. Throughout their lifetimes, applications often write a small part of each page; typically only 1-8 cache-lines out of 64 [111]. Page-granular access causes large dirty data amplification and bandwidth overuse. In contrast, fine-grained memory access over a large memory pool causes high meta-data management overhead. Based on an application's memory access patterns, remote memory abstractions should support transparent and dynamic adjustments to memory access granularity.

Memory-QoS Interface. Traditional solutions for memory page management focus on tracking (a subset of) pages and counting accesses to determine the heat of the page and then moving pages around. While this is enough to provide a two-level, hot-vs-cold QoS, it cannot capture the entire spectrum of page temperature. Potential solutions include assigning a QoS level to (1) an entire application; (2) individual data structures; (3) individual `mmap()` calls; or even (4) individual memory accesses. Each of these approaches have their pros and cons. At one extreme, assigning a QoS level to an entire application maybe simple, but it cannot capture time-varying page temperature of large, long-running applications. At the other end, assigning QoS levels to

individual memory accesses requires recompilation of all existing applications as well as cumbersome manual assignments, which can lead to erroneous QoS assignments. A combination of aforementioned approaches may reduce developer's overhead while providing sufficient flexibility to perform spatiotemporal memory QoS management.

6.1.2 Management and Runtime

Memory Address Space Management. From CXL 2.0 onward, devices can be hot-plugged to the CXL switches. Device-attached memory is mapped to the system's coherent address space and accessible to host using standard write-back semantics. Memory located on a CXL device can either be mapped as Host-managed Device Memory (HDM) or Private Device Memory (PDM). To update the memory address space for connected devices to different host devices, a system reset is needed; traffic towards the device needs to stop to alter device address mapping during this reset period. An alternate solution to avoid this system reset is to map the whole physical address space to each host when a CXL-device is added to the system. The VMM or fabric manager in the CXL switch will be responsible to maintain isolation during address-space management. How to split the whole address-space in to sizable memory blocks for the efficient physical-to-virtual address translation of a large memory network is an interesting challenge [206, 314].

Unified Runtime for Compute Disaggregation. CXL Type-2 devices (accelerator with memory) maintains cache coherency with the CPU. CPU and Type-2 devices can interchangeably use each other's memory and both get benefited. For example, applications that run on CPUs can benefit as they can now access very high bandwidth GPU memory. Similarly, for GPU users, it is beneficial for capacity expansion even though the memory bandwidth to and from CPU memory will be lower. In such a setup, remote memory abstractions should track the availability of compute cores and efficiently perform near-memory computation to improve the overall system throughput.

Future datacenters will likely be equipped with numerous domain-specific compute resources/accelerators. In such a heterogeneous system, one can borrow the idle cores of one compute resource and perform extra computation to increase the overall system throughput. A unified runtime to support malleable processes that can be immediately decomposed into smaller pieces and offloaded to any available compute nodes can improve both application and cluster throughput [266, 312].

6.1.3 Allocation Policies

Memory Allocation in Heterogenous NUMA Cluster. For better performance, hottest pages need to be on the fastest memory tier. However, due to memory capacity constraints across differ-

ent tiers, it may not always be possible to utilize the fastest or performant memory tier. Determining what fraction of memory is needed at a particular memory tier to maintain the desired performance of an application at different points of its life cycle is challenging. This is even more difficult when multiple applications coexist. Efficient promotion or demotion of pages of different temperatures across memory tiers at rack scale is necessary. One can consider augmenting TPPby incorporating a lightweight but effective algorithm to select the migration target considering node distances from the CPU, load on CPU-memory bus, current load on different memory tiers, network state, and the QoS requirements of the migration-candidate pages.

Allocation Policy for Memory Bandwidth Expansion. For memory bandwidth-bound applications, CPU-to-DRAM bandwidth often becomes the bottleneck and increases the average memory access latency. CXL's additional memory bandwidth can help by spreading memory across the top-tier and remote nodes. Instead of only placing cold pages into CXL-Memory, which has low bandwidth consumption, an ideal solution should place the right amount of bandwidth-heavy, latency-insensitive pages to CXL-Memory. The methodology to identify the ideal fraction of such working sets may even require hardware support.

Memory Sharing and Consistency. CXL 3.0 allows memory sharing across multiple devices. Through an enhanced coherency semantics, multiple hosts can have a coherent copy of a shared segment, with back invalidation for synchronization. Memory sharing improves application-level performance by reducing unnecessary data movement and improves memory utilization. Sharing a large memory address space, however, results in significant overhead and complexity in the system that plagued classic distributed shared memory (DSM) proposals [244]. Furthermore, sharing memory across multiple devices increases the security threat in the presence of any malicious application running on the same hardware space. We believe that disaggregated memory systems should cautiously approach memory sharing and avoid it unless it is absolutely necessary.

6.1.4 Rack-Level Objectives

Rack-Scale Memory Temperature. To obtain insights into an application's expected performance with multiple temperature tiers, it is necessary to understand the heat map of memory usage for that application. Existing hot page identification mechanisms (including Chameleon) are limited to a single host OS or user-space mechanism. They either use access bit-based mechanism [32, 46, 291], special CPU feature-based (e.g., Intel PEBS) tools [261, 303, 274], or OS features [60, 225] to determine the page temperature within a single server. So far, there is no distributed mechanism to determine the cluster-wide relative page temperature. Combining the data of all the

OS or user-space tools and coordinating between them to find rack-level hot pages is an important problem. CXL fabric manager is perhaps the place where one can get a cluster-wide view of hardware counters for each CXL device's load, hit, and access-related information. One can envision extending Chameleon for rack-scale environments to provide observability into each application's per-device memory temperature.

Hardware-Software Co-Design for a Better Ecosystem. Hardware features can further enhance performance of memory disaggregation systems in rack-scale setups. A memory-side cache and its associated prefetcher on the CXL ASIC or switch might help reduce the effective latency of CXL-Memory. Hardware support for data movement between memory tiers can help reduce page migration overheads in an aggressively provisioned system with very small amount of local memory and high amount of CXL-Memory. Additionally, the fabric manager of a CXL switch should implement policies like fair queuing, congestion control, load balancing etc. for better network management. Incorporating Leap's prefetcher and Hydra's erasure-coded resilience ideas into CXL switch designs can enhance system-wide performance.

Energy- and Carbon-Aware Memory Disaggregation. Datacenters represent a large and growing source of energy consumption and carbon emissions [101]. Some estimates place datacenters to be responsible for 1-2% of aggregate worldwide electricity consumption [176, 252]. To reduce the TCO and carbon footprint, and enhance hardware life expectancy, datacenter rack maintain a physical energy budget or power cap. Rack-scale memory allocation, demotion, and promotion policies can be augmented by incorporating energy-awareness in their decision-making process. In general, we can introduce energy-awareness in the software stack that manage compute, memory, and network resources in a disaggregated cluster.

6.2 Concluding Remarks

I started my PhD in 2017 with the research question in mind – *How to make memory disaggregation practical?* At that time, we had the conviction that memory disaggregation is inevitable, armed only with a few data points that hinted it might be within reach. As I conclude this dissertation in 2023, we have successfully built a comprehensive software-based disaggregated memory solution over ultra-fast RDMA networks that can provide a seamless experience for most memory-intensive applications. With diverse cache-coherent interconnects finally converging under the CXL banner, the entire industry and academia are at the cusp of taking a leap toward next-generation software-hardware co-designed disaggregated systems. Most of the hyperscalers are shifting towards a disaggregated datacenter design to reduce the TCO and improve the effective application-level

performance. The software solutions presented in this dissertation are the stepping stones for the practical usability for memory disaggregation systems. We are happy that our solutions have a wide acceptance in the industry and already been deployed in hyperscaler datacenters and merged to Linux mainstream. We envision, many more influential research in this field to build a software-hardware ecosystem supporting memory disaggregation.

BIBLIOGRAPHY

[1] A Milestone in Moving Data. https://newsroom.intel.com/editorials/milestone-moving-data/.

[2] A Twitter Analog to PageRank. http://thenoisychannel.com/2009/01/13/a-twitter-analog-to-pagerank.

[3] Accelio based network block device. https://github.com/accelio/NBDX.

[4] Alibaba Cluster Trace 2018. https://github.com/alibaba/clusterdata/blob/master/cluster-trace-v2018/trace_2018.md.

[5] Amazon Aurora Serverless. https://aws.amazon.com/rds/aurora/serverless/.

[6] Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/.

[7] Amazon EC2 Pricing. https://aws.amazon.com/ec2/pricing. Accessed: 2019-08-05.

[8] Amazon EC2 Pricing. https://aws.amazon.com/ec2/pricing/on-demand/.

[9] Amazon Elastic Block Store. https://aws.amazon.com/ebs/.

[10] Amazon ElastiCache. https://aws.amazon.com/elasticache/.

[11] AMD Infinity Architecture. https://www.amd.com/en/technologies/infinity-architecture.

[12] AMD Joins Consortia to Advance CXL. https://community.amd.com/t5/amd-business-blog/amd-joins-consortia-to-advance-cxl-a-new-high-speed-interconnect/ba-p/418202.

[13] Apache Hadoop NextGen MapReduce (YARN). http://goo.gl/etTGA.

[14] Apache Spark: Lightning-Fast Unified Analytics Engine. https://spark.apache.org/.

[15] AWS Lambda. https://aws.amazon.com/lambda/.

[16] AWS Lambda Limits. https://docs.aws.amazon.com/lambda/latest/dg/limits.html.

[17] AWS Spot Instance Pricing History. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-spot-instances-history.html.

[18] AWS Transit Gateway. https://aws.amazon.com/transit-gateway.

[19] Azure Cache for Redis. https://azure.microsoft.com/en-us/services/cache/.

[20] Azure Functions. https://azure.microsoft.com/en-us/services/functions/.

[21] Baidu feed stream services restructures its in-memory database with intel optane technology. https://www.intel.com/content/www/us/en/customer-spotlight/stories/baidu-feed-stream-case-study.html.

[22] CCIX. https://www.ccixconsortium.com/.

[23] Cgroups. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[24] Classification for Biospecies 3. https://www.kaggle.com/olgabelitskaya/tf-cats-vs-dogs/version/2.

[25] CloudLab. https://www.cloudlab.us/.

[26] ApsaraDB for POLARDB: A next-generationrelational database - alibaba cloud. https://www.alibabacloud.com/products/apsaradb-for-polardb .

[27] Compute Express Link (CXL). https://www.computeexpresslink.org/.

[28] Creating in-memory RAM disks. https://cloud.google.com/compute/docs/disks/mount-ram-disks.

[29] Custom Machine Types. https://cloud.google.com/custom-machine-types/.

[30] CXL and the Tiered-Memory Future of Servers. shorturl.at/ghwL7.

[31] CXL Roadmap Opens Up the Memory Hierarchy. https://www.nextplatform.com/2021/09/07/the-cxl-roadmap-opens-up-the-memory-hierarchy/.

[32] DAMON: Data Access MONitoring Framework for Fun and Memory Management Optimizations. https://www.linuxplumbersconf.org/event/7/contributions/659/attachments/503/1195/damon_ksummit_2020.pdf.

[33] etcd. https://github.com/etcd-io/etcd.

[34] Facebook and Amazon are causing a memory shortage. https://www.networkworld.com/article/3247775/facebook-and-amazon-are-causing-a-memory-shortage.html.

[35] Facebook announces next-generation Open Rack frame. https://engineering.fb.com/2019/03/15/data-center-engineering/open-rack/.

[36] Fio - Flexible I/O Tester. https://github.com/axboe/fio.

[37] Frontswap. https://www.kernel.org/doc/html/latest/vm/frontswap.html.

[38] Gen-Z. https://genzconsortium.org/.

[39] Global VNet Peering now generally available? https://azure.microsoft.com/en-us/blog/global-vnet-peering-now-generally-available/.

[40] Google Cloud Networking Incident 20005. https://status.cloud.google.com/incident/cloud-networking/20005.

[41] Google Cluster Trace 2019. https://github.com/google/cluster-data/blob/master/ClusterData2019.md.

[42] Google Compute Engine Pricing. https://cloud.google.com/compute/pricing. Accessed: 2019-08-05.

[43] Graph Analytics Benchmark in CloudSuite. http://parsa.epfl.ch/cloudsuite/graph.html.

[44] HP: The Machine. http://www.labs.hpe.com/research/themachine/.

[45] Idle Memory Tracking. https://www.kernel.org/doc/Documentation/vm/idle_page_tracking.txt.

[46] Idle page tracking-based working set estimation. https://lwn.net/Articles/460762/.

[47] Infiniswap github repository. https://github.com/SymbioticLab/infiniswap.

[48] Intel Intelligent Storage Acceleration Library (Intel ISA-L). https://software.intel.com/en-us/storage/ISA-L.

[49] Intel Rack Scale Design Architecture Overview. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/rack-scale-design-architecture-white-paper.pdf.

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

[51] Introducing new product innovations for SAP HANA, Expanded AI collaboration with SAP and more. https://azure.microsoft.com/en-us/blog/introducing-new-product-innovations-for-sap-hana-expanded-ai-collaborat

[52] Kubernetes. http://kubernetes.io.

[53] Linux Virtual Machines Pricing. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/.

[54] Mellanox InfiniBand Adapter Cards. https://www.mellanoxstore.com/categories/adapters/infiniband-and-vpi-adapter-cards.html.

[55] Mellanox Switches. https://www.mellanoxstore.com/categories/switches/infiniband-and-vpi-switch-systems.html.

[56] Memcached - A distributed memory object caching system. http://memcached.org.

[57] MemCachier. https://www.memcachier.com/.

[58] Memkind. https://memkind.github.io/memkind/.

[59] Micron Exits 3DXPoint, Eyes CXL Opportunities. https://www.eetimes.com/micron-exits-3d-xpoint-market-eyes-cxl-opportunities.

[60] NUMA Balancing (AutoNUMA). https://mirrors.edge.kernel.org/pub/linux/kernel/people/andrea/autonuma/autonuma_bench-20120530.pdf.

[61] Open Compute Project : Open Rack Charter. https://github.com/facebookarchive/opencompute/blob/master/open_rack/charter/Open_Rack_Charter.pdf.

[62] OpenCAPI. https://opencapi.org/.

[63] Preemptible VM Instances. https://cloud.google.com/compute/docs/instances/preemptible.

[64] Presto. https://prestodb.io.

[65] Pricing of Intel's Optane DC Persistent Memory. https://www.anandtech.com/show/14180/pricing-of-intels-optane-dc-persistent-memory-modules-leaks.

[66] Rack-scale computing at Yahoo! http://www.intel.com/content/dam/www/public/us/en/documents/presentation/idf15-yahoo-rack-scale-computing-presentation.pdf.

[67] Redis, an in-memory data structure store. http://redis.io.

[68] Redis Labs. https://redislabs.com/.

[69] Reimagining Memory Expansion for Single Socket Servers with CXL. https://www.computeexpresslink.org/post/cxl-consortium-upcoming-industry-events.

[70] Samsung Unveils Industry-First Memory Module Incorporating New CXL Interconnect Standard. https://news.samsung.com/global/samsung-unveils-industry-first-memory-module-incorporating-new-cxl-inte

[71] Spot Instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-spot-instances.html.

[72] Storm: Distributed and fault-tolerant realtime computation. http://storm-project.net.

[73] Tencent explores datacenter resource-pooling using Intel rack scale architecture (Intel RSA). http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/rsa-tencent-paper.pdf.

[74] TensorFlow - An end-to-end open source machine learning platform. https://www.tensorflow.org/.

[75] The Versatile SMP (vSMP) Architecture. http://www.scalemp.com/technology/versatile-smp-vsmp-architecture/.

[76] The zswap compressed swap cache. https://lwn.net/Articles/537422/.

[77] Tmpfs. https://www.kernel.org/doc/html/latest/filesystems/tmpfs.html.

[78] Top-tier memory management. https://lwn.net/Articles/857133/.

[79] TPC Benchmark C (TPC-C). http://www.tpc.org/tpcc/.

[80] Use low-priority VMs with Batch. https://docs.microsoft.com/en-us/azure/batch/batch-low-pri-vms.

[81] Using DAMON for proactive reclaim. https://lwn.net/Articles/863753/.

[82] Vertical Pod Autoscaler. https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler.

[83] VM instances pricing. https://cloud.google.com/compute/vm-instance-pricing.

[84] VoltDB. https://github.com/VoltDB/voltdb.

[85] What is VPC peering? https://docs.aws.amazon.com/vpc/latest/peering/what-is-vpc-peering.html.

[86] Who's using Redis? https://redis.io/topics/whos-using-redis.

[87] zram. https://www.kernel.org/doc/Documentation/blockdev/zram.txt.

[88] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. *SIGPLAN*, 2017.

[89] Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafrir. Deconstructing Amazon EC2 spot instance pricing. *ACM Transactions on Economics and Computation*, 2013.

[90] Orna Agmon Ben-Yehuda, Eyal Posener, Muli Ben-Yehuda, Assaf Schuster, and Ahuva Mu'alem. Ginseng: Market-driven memory allocation. *SIGPLAN*, 2014.

[91] M. K. Aguilera, R. Janakiraman, and L. Xu. Using erasure codes efficiently for storage in a distributed system. In *DSN*, 2005.

[92] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote regions: a simple abstraction for remote memory. In *USENIX ATC*, 2018.

[93] Marcos K Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote memory in the age of fast networks. In *SoCC*, 2017.

[94] Marcos K Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote memory in the age of fast networks. In *SoCC*, 2017.

[95] Ameen Akel, Adrian M. Caulfield, Todor I. Mollov, Rajesh K. Gupta, and Steven Swanson. Onyx: A protoype phase change memory storage array. In *HotStorage*, 2011.

[96] S. Alamro, M. Xu, T. Lan, and S. Subramaniam. CRED: Cloud right-sizing to meet execution deadlines and data locality. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, 2016.

[97] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. CherryPick: Adaptively unearthing the best cloud configurations for big data analytics. In *NSDI*, 2017.

[98] 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.

[99] 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.

[100] 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 *OSDI*, 2020.

[101] Thomas Anderson, Adam Belay, Mosharaf Chowdhury, Asaf Cidon, and Irene Zhang. Treehouse: A case for carbon-aware datacenter software. In *HotCarbon*, 2022.

[102] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational data processing in Spark. In *SIGMOD*, 2015.

[103] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. *SIGMETRICS Perform. Eval. Rev.*, 2012.

[104] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS*, 2012.

[105] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS*, 2012.

[106] Jean-Loup Baer and Tien-Fu Chen. An effective on-chip preloading scheme to reduce data access penalty. In *ACM/IEEE Conference on Supercomputing*, 1991.

[107] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. Rack-scale in-memory join processing using RDMA. In *SIGMOD*, 2015.

[108] Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafrir. The Resource-as-a-Service (RaaS) cloud. In *HotCloud*, 2012.

[109] 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 *OSDI*, 2020.

[110] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux block io: Introducing multi-queue ssd access on multi-core systems. In *SYSTOR*, 2013.

[111] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking software runtimes for disaggregated memory. In *ASPLOS*, 2021.

[112] Irina Calciu, Ivan Puddu, Aasheesh Kolli, Andreas Nowatzyk, Jayneel Gandhi, Onur Mutlu, and Pratap Subrahmanyam. Project pberry: Fpga acceleration for remote memory. HotOS, 2019.

[113] Pei Cao, Edward W. Felten, and Kai Li. Implementation and performance of application-controlled file caching. In *OSDI*, 1994.

[114] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *FAST*, 2020.

[115] Amanda Carbonari and Ivan Beschasnikh. Tolerating faults in disaggregated datacenters. In *HotNets*, 2017.

[116] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2015.

[117] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *MICRO*, 2010.

[118] Jeremy C. W. Chan, Qian Ding, Patrick P. C. Lee, and Helen H. W. Chan. Parity logging with reserved space: Towards efficient updates and recovery in erasure-coded clustered storage. In *FAST*, 2014.

[119] Haibo Chen, Heng Zhang, Mingkai Dong, Zhaoguo Wang, Yubin Xia, Haibing Guan, and Binyu Zang. Efficient and available in-memory kv-store with hybrid erasure coding and replication. *ACM Transactions on Storage (TOS)*, 13(3):25, 2017.

[120] Haogang Chen, Yingwei Luo, Xiaolin Wang, Binbin Zhang, Yifeng Sun, and Zhenlin Wang. A transparent remote paging model for virtual machines. In *International Workshop on Virtualization Technology*, 2008.

[121] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *SIGKDD*, 2016.

[122] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

[123] Yu Chen, Ivy B. Peng, Zhen Peng, Xu Liu, and Bin Ren. ATMem: Adaptive data placement in graph applications on heterogeneous memories. In *CGO*, 2020.

[124] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, et al. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *IPDPSW*, 2016.

[125] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Dynacache: Dynamic cloud caching. In *HotCloud*, 2015.

[126] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *NSDI*, 2016.

[127] Asaf Cidon, Robert Escriva, Sachin Katti, Mendel Rosenblum, and Emin Gun Sirer. Tiered replication: A cost-effective alternative to full cluster geo-replication. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 31–43, Santa Clara, CA, July 2015. USENIX Association.

[128] Asaf Cidon, Stephen M Rumble, Ryan Stutsman, Sachin Katti, John Ousterhout, and Mendel Rosenblum. Copysets: Reducing the frequency of data loss in cloud storage. In *USENIX ATC*, 2013.

[129] Asaf Cidon, Daniel Rushton, Stephen M. Rumble, and Ryan Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *USENIX ATC*, 2017.

[130] Edward G. Coffman, Jr. and Peter J. Denning. *Operating Systems Theory*. Prentice Hall Professional Technical Reference, 1973.

[131] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R. Narasayya. Accelerating relational databases by leveraging remote memory and RDMA . In *SIGMOD*, 2016.

[132] Jeffrey Dean. Evolution and future directions of large-scale storage and computation systems at google . In *SoCC*, 2010.

[133] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane DC persistent memory module . *arXiv preprint arXiv:1903.05714*, 2019.

[134] K. Katrinis, D. Syrivelis, D. Pnevmatikatos, G. Zervas, D. Theodoropoulos, I. Koutsopoulos, K. Hasharoni, D. Raho, C. Pinto, F. Espina, S. Lopez-Buedo, Q. Chen, M. Nemirovsky, D. Roca, H. Klos, and T. Berends. Rack-scale disaggregated cloud data centers: The dReDBox project vision . In *DATE*, 2016.

[135] Krste Asanović. FireBox: A hardware building block for 2020 warehouse-scale computers . In *FAST*. USENIX Association, 2014.

[136] Robert J. Chansler. Data availability and durability with the hadoop distributed file system . *login Usenix Mag.*, 37, 2012.

[137] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, 2010.

[138] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 153–167. ACM, 2017.

[139] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[140] Peter J. Denning. The working set model for program behavior. *Commun. ACM*, 11(5):323–333, May 1968.

[141] Thaleia Dimitra Doudali, Sergey Blagodurov, Abhinav Vishnu, Sudhanva Gurumurthi, and Ada Gavrilovska. Kleio: A hybrid memory page scheduler with machine intelligence. In *HPDC*, 2019.

[142] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast remote memory. In *NSDI*, 2014.

[143] Junping Du and Yue Li. Elastify cloud-native spark application with PMEM. Persistent Memory Summit, 2019.

[144] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *EuroSys*, 2016.

[145] Sandhya Dwarkadas, Nikolaos Hardavellas, Leonidas Kontothanassis, Rishiyur Nikhil, and Robert Stets. Cashmere-VLM: Remote memory paging for software distributed shared memory. In *IPPS/SPDP*, 1999.

[146] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing DRAM footprint with NVM in Facebook. In *EuroSys*, 2018.

[147] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim Hazelwood, Asaf Cidon, and Sachin Katti. Bandana: Using non-volatile memory for storing deep learning models. *SysML*, 2019.

[148] Viacheslav Fedorov, Jinchun Kim, Mian Qin, Paul V. Gratz, and A. L. Narasimha Reddy. Speculative paging for future NVM storage. In *MEMSYS*, 2017.

[149] Michael J Feeley, William E Morgan, EP Pighin, Anna R Karlin, Henry M Levy, and Chandramohan A Thekkath. Implementing global memory management in a workstation cluster. In *ACM SIGOPS Operating Systems Review*, volume 29, pages 201–212. ACM, 1995.

[150] Michael J Feeley, William E Morgan, EP Pighin, Anna R Karlin, Henry M Levy, and Chandramohan A Thekkath. Implementing global memory management in a workstation cluster. In *SOSP*, 1995.

[151] Edward W. Felten and John Zahorjan. Issues in the implementation of a remote memory paging system. Technical Report 91-03-09, University of Washington, Mar 1991.

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

[153] Michail D. Flouris and Evangelos P. Markatos. The network RamDisk: Using remote memory on heterogeneous NOWs. *Journal of Cluster Computing*, 2(4):281–293, 1999.

[154] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in globally distributed storage systems. In *OSDI*, 2010.

[155] 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 *ASPLOS*, 2022.

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

[157] 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*, 2016.

[158] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiaji Zhu, Haiyong Wang, Dennis Cai, and Jiesheng Wu. When cloud storage meets RDMA. In *NSDI*, 2021.

[159] Ahmad Ghalayini, Jinkun Geng, Vighnesh Sachidananda, Vinay Sriram, Yilong Geng, Balaji Prabhakar, Mendel Rosenblum, and Anirudh Sivaraman. CloudEx: A fair-access financial exchange in the cloud. In *HotOS*, 2021.

[160] Amol Ghoting, Rajasekar Krishnamurthy, Edwin Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. SystemML: Declarative machine learning on MapReduce. In *ICDE*, 2011.

[161] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Power-Graph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[162] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[163] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct access, High-Performance memory disaggregation with DirectCXL. In *USENIX ATC*, 2022.

[164] James Griffioen and Randy Appleton. Reducing file system latency using a predictive approach. In *USTC*, 1994.

[165] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with Infiniswap. In *NSDI*, 2017.

[166] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. Network support for resource disaggregation in next-generation datacenters. In *HotNets*, 2013.

[167] Dave Hansen. Migrate pages in lieu of discard. https://lwn.net/Articles/860215/.

[168] S. L. Ho and M. Xie. The use of ARIMA models for reliability forecasting and analysis. *Comput. Ind. Eng.*, 1998.

[169] Brian Holden, Don Anderson, Jay Trodden, and Maryanne Daves. *HyperTransport 3.1 Interconnect Technology*. 2008.

[170] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Chen Ding, and Zhenlin Wang. Kinetic modeling of data eviction in cache. In *USENIX ATC*, 2016.

[171] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in Windows Azure Storage. In *USENIX ATC*, 2012.

[172] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *USENIX ATC*, 2010.

[173] Akanksha Jain and Calvin Lin. Linearizing irregular memory accesses for improved correlated prefetching. In *MICRO*, 2013.

[174] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *ACM MM*, 2014.

[175] Song Jiang, Xiaoning Ding, Feng Chen, Enhua Tan, and Xiaodong Zhang. Dulo: An effective buffer cache management scheme to exploit both temporal and spatial localities. In *FAST*, 2005.

[176] Nicola Jones. How to stop data centres from gobbling up the world's electricity. *Nature*, 561:163–166, 2018.

[177] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *ISCA*, 1997.

[178] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using RDMA efficiently for key-value services. In *SIGCOMM*, 2014.

[179] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance RDMA systems. In *ATC*, 2016.

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

[181] Anuj Kalia Michael Kaminsky and David G Andersen. Using rdma efficiently for key-value services. In *SIGCOMM*, 2014.

[182] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The nature of datacenter traffic: Measurements and analysis. In *IMC*, 2009.

[183] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. Heteroos: Os design for heterogeneous memory management in datacenter. 2017.

[184] Scott F. Kaplan, Lyle A. McGeoch, and Megan F. Cole. Adaptive caching for demand prepaging. *SIGPLAN Not.*, 2002.

[185] Hiwot Tadese Kassa, Jason Akers, Mrinmoy Ghosh, Zhichao Cao, Vaibhav Gogte, and Ronald Dreslinski. Improving performance of flash based Key-Value stores using storage class memory as a volatile memory extension. In *USENIX ATC*, 2021.

[186] K. Katrinis, D. Syrivelis, D. Pnevmatikatos, G. Zervas, D. Theodoropoulos, I. Koutsopoulos, K. Hasharoni, D. Raho, C. Pinto, F. Espina, S. Lopez-Buedo, Q. Chen, M. Nemirovsky, D. Roca, H. Klos, and T. Berends. Rack-scale disaggregated cloud data centers: The dReDBox project vision. In *DATE*, 2016.

[187] Srinivasan Keshav and S Kesahv. *An engineering approach to computer networking: ATM networks, the Internet, and the telephone network*, volume 116. Addison-Wesley Reading, 1997.

[188] M. Khan, A. Sandberg, and E. Hagersten. A case for resource efficient prefetching in multicores. In *ICPP*, 2014.

[189] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. Exploring the design space of page management for Multi-Tiered memory systems. In *USENIX ATC*, 2021.

[190] Ana Klimovic and Christos Kozyrakis. ReFlex : Remote Flash = Local Flash. In *ASPLOS*, 2017.

[191] Ana Klimovic, Christos Kozyrakis, and Binu John. Flash Storage Disaggregation. In *EuroSys*, 2016.

[192] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Selecta: Heterogeneous cloud storage configuration for data analytics. In *USENIX ATC*, 2018.

[193] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *OSDI*, 2018.

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

[195] Marcel Kornacker, A Behm, V Bittorf, T Bobrovytsky, C Ching, A Choi, J Erickson, M Grund, D Hecht, M Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. Impala: A modern, open-source SQL engine for Hadoop. In *CIDR*, 2015.

[196] Tim Kraska, Ameet Talwalkar, John C Duchi, Rean Griffith, Michael J Franklin, and Michael I Jordan. MLbase: A distributed machine-learning system. In *CIDR*, 2013.

[197] Chinmay Kulkarni, Aniraj Kesavan, Tian Zhang, Robert Ricci, and Ryan Stutsman. Rocksteady: Fast migration for low-latency in-memory storage. In *SOSP*, 2017.

[198] Yossi Kuperman, Joel Nider, Abel Gordon, and Dan Tsafrir. Paravirtual Remote I/O. In *ASPLOS*, 2016.

[199] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *WWW*, 2010.

[200] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *WWW*, 2010.

[201] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with Ingens. In *OSDI*, 2016.

[202] 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*, 2019.

[203] 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*, 2019.

[204] Collin Lee and John Ousterhout. Granular computing. In *HotOS*, 2019.

[205] Seok-Hee Lee. Technology scaling challenges and opportunities of memory devices. In *2016 IEEE International Electron Devices Meeting (IEDM)*, 2016.

[206] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. MIND: In-network memory management for disaggregated data centers. In *SOSP*, 2021.

[207] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G. Shin. Mitigating the performance-efficiency tradeoff in resilient memory disaggregation. *arXiv preprint arXiv:1910.09727*, 2020.

[208] Huaicheng Li, Daniel S. Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: CXL-based memory pooling systems for cloud platforms. In *ASPLOS*, 2023.

[209] Yang Li, Saugata Ghose, Jongmoo Choi, Jin Sun, Hui Wang, and Onur Mutlu. Utility-based hybrid memory management. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, 2017.

[210] S. Liang, R. Noronha, and D. K. Panda. Swapping to remote memory over infiniband: An approach using a high performance network block device. In *IEEE International Conference on Cluster Computing*, 2005.

[211] Shuang Liang, Ranjit Noronha, and Dhabaleswar K Panda. Swapping to remote memory over Infiniband: An approach using a high performance network block device. In *Cluster Computing*, 2005.

[212] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *NSDI*, 2014.

[213] K. Lim, Y. Turner, Jichuan Chang, J. Santos, and P. Ranganathan. Disaggregated memory benefits for server consolidation. 2011.

[214] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. *SIGARCH*, 2009.

[215] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K Reinhardt, and Thomas F Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *ISCA*, 2009.

[216] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F Wenisch. System-level implications of disaggregated memory. In *HPCA*, 2012.

[217] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. GraphLab: A new framework for parallel machine learning. In *UAI*, 2010.

[218] Xiaoyi Lu, Nusrat S. Islam, Md. Wasi-Ur-Rahman, Jithin Jose, Hari Subramoni, Hao Wang, and Dhabaleswar K. Panda. High-performance design of hadoop rpc with rdma over infiniband. In *ICPP*, 2013.

[219] Chris A. Mack. Fifty years of moore's law. *IEEE Transactions on Semiconductor Manufacturing*, 2011.

[220] Dan Magenheimer. Transcendent memory on Xen. *Xen Summit*, 2009.

[221] Dan Magenheimer, Chris Mason, Dave McCracken, and Kurt Hackel. Paravirtualized paging. In *WIOV*, 2008.

[222] Dan Magenheimer, Chris Mason, Dave McCracken, and Kurt Hackel. Transcendent memory and linux. In *Proceedings of the Linux Symposium*, pages 191–200. Citeseer, 2009.

[223] Evangelos P Markatos and George Dramitinos. Implementation of a reliable remote memory pager. In *USENIX ATC*, 1996.

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

[225] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. TPP: Transparent page placement for CXL-enabled tiered-memory. In *ASPLOS*, 2023.

[226] Hasan Al Maruf, Yuhong Zhong, Hongyi Wong, Mosharaf Chowdhury, Asaf Cidon, and Carl Waldspurger. Memtrade: A disaggregated-memory marketplace for public clouds. *arXiv preprint arXiv:2108.06893*, 2021.

[227] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Trans. Comput. Syst.*, 1984.

[228] Xiangrui Meng, Joseph K. Bradley, Burak Yavuz, Evan R. Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D. B. Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. MLlib: Machine learning in Apache Spark. *CoRR*, abs/1505.06807, 2015.

[229] Mitesh R. Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H. Loh. Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories. In *HPCA*, 2015.

[230] Vaibhawa Mishra, Joshua L. Benjamin, and Georgios Zervas. MONet: heterogeneous memory over optical network for large-scale data center resource disaggregation. *Journal of Optical Communications and Networking*, 2021.

[231] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *USENIX ATC*, 2013.

[232] Sparsh Mittal. A survey of recent prefetching techniques for processor caches. *ACM Comput. Surv.*, 2016.

[233] Michael Mitzenmacher, Andrea W. Richa, and Ramesh Sitaraman. The power of two random choices: A survey of techniques and results. *Handbook of Randomized Computing*, pages 255–312, 2001.

[234] Jeffrey C. Mogul, Eduardo Argollo, Mehul Shah, and Paolo Faraboschi. Operating system support for nvm+dram hybrid main memory. In *HotOS*, 2009.

[235] Jeffrey C Mogul and John Wilkes. Nines are not enough: Meaningful metrics for clouds. In *HotOS*, 2019.

[236] Ingo Müller, Rodrigo FBP Bruno, Ana Klimovic, Gustavo Alonso, John Wilkes, and Eric Sedlar. Serverless clusters: The missing piece for interactive batch applications? In *SPMA*, 2020.

[237] Subramanian Muralidhar, Wyatt Lloyd, Southern California, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, and Sanjeev Kumar. Facebook's Warm BLOB Storage System. In *OSDI*, 2014.

[238] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. Naiad: A timely dataflow system. In *SOSP*, 2013.

[239] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *USENIX ATC*, 2015.

[240] K.J. Nesbit and J.E. Smith. Data cache prefetching using a global history buffer. *IEEE Micro*, 2005.

[241] Tia Newhall, Sean Finney, Kuzman Ganchev, and Michael Spiegel. Nswap: A network swapping module for Linux clusters. In *Euro-Par*, 2003.

[242] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at Facebook. In *NSDI*, 2013.

[243] Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, and Daniel Hagimont. Welcome to zombieland: Practical and energy-efficient memory disaggregation in a datacenter. In *EuroSys*, 2018.

[244] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, 1991.

[245] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. Scale-out NUMA. In *ASPLOS*, 2014.

[246] Travis Oliphant. NumPy: A guide to NumPy. USA: Trelgol Publishing, 2006.

[247] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC*, 2014.

[248] Diego Ongaro, Stephen M Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMCloud. In *SOSP*, 2011.

[249] Mark Oskin and Gabriel H. Loh. A software-managed approach to die-stacked dram. In *PACT*, 2015.

[250] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for RAMClouds: Scalable high performance storage entirely in DRAM. *SIGOPS OSR*, 43(4), 2010.

[251] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for RAMClouds: Scalable high-performance storage entirely in DRAM. *SIGOPS Oper. Syst. Rev.*, 2010.

[252] Fred Pearce. Energy hogs: Can world's huge data centers be made more efficient? *Yale Environment*, 2018.

[253] Leeor Peled, Shie Mannor, Uri Weiser, and Yoav Etsion. Semantic locality and context-based prefetching using reinforcement learning. In *ISCA*, 2015.

[254] Russell Power and Jinyang Li. Building fast, distributed programs with partitioned tables. In *OSDI*, 2010.

[255] Konstantinos Psounis, Balaji Prabhakar, and Dawson Engler. A randomized cache replacement scheme approximating LRU. In *Proceedings of the 34th Annual Conference on Information Sciences and Systems*, March 2000.

[256] Rodric M. Rabbah, Hariharan Sandanagobalane, Mongkol Ekpanyapong, and Weng-Fai Wong. Compiler orchestrated prefetching via speculation and predication. In *ASPLOS*, 2004.

[257] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page placement in hybrid memory systems. In *ICS*, 2011.

[258] Pramod Subba Rao and George Porter. Is memory disaggregation feasible?: A case study with Spark SQL. In *ANCS*, 2016.

[259] K V Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. EC-Cache: Load-balanced, low-latency cluster caching with online erasure coding. In *OSDI*, 2016.

[260] KV Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. EC-Cache: Load-balanced, low-latency cluster caching with online erasure coding. In *OSDI*, 2016.

[261] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. *HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM*. 2021.

[262] I.S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[263] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *SoCC*, 2012.

[264] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *SoCC*, 2012.

[265] Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann. High-speed query processing over high-speed networks. In *VLDB*, 2015.

[266] Zhenyuan Ruan, Seo Jin Park, Marcos K. Aguilera, Adam Belay, and Malte Schwarzkopf. Nu: Achieving microsecond-scale resource fungibility with logical processes. In *NSDI*, 2023.

[267] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-performance, application-integrated far memory. In *OSDI*, 2020.

[268] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. Autopilot: Workload autoscaling at Google. In *EuroSys*, 2020.

[269] Tudor-Ioan Salomie, Gustavo Alonso, Timothy Roscoe, and Kevin Elphinstone. Application level ballooning for efficient server consolidation. In *EuroSys*, 2013.

[270] Ahmad Samih, Ren Wang, Christian Maciocco, Tsung-Yuan Charlie Tai, Ronghui Duan, Jiangang Duan, and Yan Solihin. Evaluating dynamics and bottlenecks of memory collaboration in cluster systems. In *CCGrid*, 2012.

[271] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris S. Papailiopoulos, Alexandros G. Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. XORing elephants: Novel erasure codes for big data. In *VLDB*, 2013.

[272] Joel H Schopp, Keir Fraser, and Martine J Silbermann. Resizing memory with balloons and hotplug. In *Proceedings of the Linux Symposium*, volume 2, pages 313–319, 2006.

[273] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In *EuroSys*, 2013.

[274] Harald Servat, Antonio J. Peña, Germán Llort, Estanislao Mercadal, Hans-Christian Hoppe, and Jesús Labarta. Automating the application data placement in hybrid memory systems. In *IEEE International Conference on Cluster Computing*, 2017.

[275] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *OSDI*, 2018.

[276] Prateek Sharma, Ahmed Ali-Eldin, and Prashant Shenoy. Resource deflation: A new approach for transient resource reclamation. In *EuroSys*, 2019.

[277] Prateek Sharma, Purushottam Kulkarni, and Prashant Shenoy. Per-VM page cache partitioning for cloud computing platforms. In *COMSNETS*, 2016.

[278] Supreeth Shastri, Amr Rizk, and David Irwin. Transient guarantees: Maximizing the value of idle cloud capacity. In *SC*, 2016.

[279] Timothy Sherwood, Suleyman Sair, and Brad Calder. Predictor-directed stream buffers. In *MICRO*, 2000.

[280] Manjunath Shevgoor, Sahil Koladiya, Rajeev Balasubramonian, Chris Wilkerson, Seth H. Pugsley, and Zeshan Chishti. Efficiently prefetching complex address patterns. In *MICRO*, 2015.

[281] Stephen Somogyi, Thomas F. Wenisch, Anastasia Ailamaki, and Babak Falsafi. Spatio-temporal memory streaming. In *ISCA*, 2009.

[282] Yang Song, Murtaza Zafer, and Kang-Won Lee. Optimal bidding in spot instance market. In *2012 Proceedings IEEE Infocom*, pages 190–198. IEEE, 2012.

[283] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *HPCA*, 2007.

[284] Akshitha Sriraman and Abhishek Dhanotia. *Accelerometer: Understanding Acceleration Opportunities for Data Center Overheads at Hyperscale*. 2020.

[285] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F. Wenisch. SoftSKU: Optimizing server architectures for microservice diversity @scale. In *ISCA*, 2019.

[286] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer attacks over the network and defenses. In *ATC*, 2018.

[287] Shin-Yeh Tsai, Mathias Payer, and Yiying Zhang. Pythia: Remote oracles for the masses. In *USENIX Security*, 2019.

[288] Shin-Yeh Tsai and Yiying Zhang. Lite kernel rdma support for datacenter applications. In *SOSP*, 2017.

[289] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon Aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1041–1052, 2017.

[290] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with Borg. In *EuroSys*, 2015.

[291] Vladimir Davydov. Idle Memory Tracking. https://lwn.net/Articles/639341/.

[292] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building an elastic query engine on disaggregated storage. In *NSDI*, 2020.

[293] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using miniature simulations. In *USENIX ATC*, 2017.

[294] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *OSDI*, 2002.

[295] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC construction with SHARDS. In *FAST*, 2015.

[296] Cheng Wang, Bhuvan Urgaonkar, Aayush Gupta, George Kesidis, and Qianlin Liang. Exploiting spot and burstable instances for improving the cost-efficacy of in-memory caches on the public cloud. In *EuroSys*, 2017.

[297] 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 *OSDI*, 2020.

[298] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J. Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. SmartHarvest: Harvesting idle CPUs safely and efficiently in the cloud. In *EuroSys*, 2021.

[299] Wei Wei, Dejun Jiang, Sally A. McKee, Jin Xiong, and Mingyu Chen. Exploiting program semantics to place data in hybrid memory. In *PACT*, 2015.

[300] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *OSDI*, 2006.

[301] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. TMO: Transparent memory offloading in datacenters. In *ASPLOS*, 2022.

[302] Yair Wiseman, Song Jiang, Yair Wiseman, and Song Jiang. *Advanced Operating Systems and Kernel Applications: Techniques and Technologies*. Information Science Reference - Imprint of: IGI Publishing, 2009.

[303] Kai Wu, Yingchao Huang, and Dong Li. Unimem: Runtime data managementon non-volatile memory-based heterogeneous main memory. In *SC*, 2017.

[304] Kai Wu, Jie Ren, and Dong Li. Runtime data management on non-volatile memory-based heterogeneous memory for task-parallel programs. In *SC*, 2018.

[305] Hong Xu and Baochun Li. Dynamic cloud pricing for revenue maximization. *IEEE Transactions on Cloud Computing*, 1(2):158–171, 2013.

[306] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble page management for tiered memory systems. In *ASPLOS*, 2019.

[307] Jisoo Yang, Dave B. Minturn, and Frank Hady. When poll is better than interrupt. In *FAST*, 2012.

[308] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *OSDI*, 2020.

[309] 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*, 2015.

[310] Matt M. T. Yiu, Helen H. W. Chan, and Patrick P. C. Lee. Erasure coding for small objects in in-memory kv storage. In *SYSTOR*, 2017.

[311] Dong Young Yoon, Mosharaf Chowdhury, and Barzan Mozafari. Distributed lock management with rdma: decentralization without starvation. In *SIGMOD*, 2018.

[312] Jie You, Jingfeng Wu, Xin Jin, and Mosharaf Chowdhury. Ship compute or ship data? why not both? In *NSDI*, 2021.

[313] Young Jin Yu, Dong In Shin, Woong Shin, Nae Young Song, Jae Woo Choi, Hyeong Seog Kim, Hyeonsang Eom, and Heon Young Yeom. Optimizing the block i/o subsystem for fast storage devices. *ACM Trans. Comput. Syst.*, 2014.

[314] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. NetLock: Fast, centralized lock management using programmable switches. In *SIGCOMM*, 2020.

[315] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant stream computation at scale. In *SOSP*, 2013.

[316] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX NSDI*, 2012.

[317] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. The end of a myth: Distributed transactions can scale. In *VLDB*, 2017.

[318] Lei Zhang, Reza Karimi, Irfan Ahmad, and Ymir Vigfusson. Optimal data placement for heterogeneous cache, memory, and storage systems. 2020.

[319] Qi Zhang, Mohamed Faten Zhani, Shuo Zhang, Quanyan Zhu, Raouf Boutaba, and Joseph L Hellerstein. Dynamic energy-aware capacity provisioning for cloud computing environments. In *ICAC*, 2012.

[320] Qi Zhang, Quanyan Zhu, and Raouf Boutaba. Dynamic resource allocation for spot markets in cloud computing environments. In *2011 Fourth IEEE International Conference on Utility and Cloud Computing*, pages 178–185. IEEE, 2011.

[321] Qizhen Zhang, Yifan Cai, Xinyi Chen, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. Understanding the effect of data center resource disaggregation on production dbmss. In *VLDB*, 2020.

[322] Y. Zhang, J. Gu, Y. Lee, M. Chowdhury, and K. G. Shin. Performance isolation anomalies in RDMA. In *ACM SIGCOMM KBNets*, 2017.

[323] Yiwen Zhang, Juncheng Gu, Youngmoon Lee, Mosharaf Chowdhury, and Kang G. Shin. Performance Isolation Anomalies in RDMA. In *KBNets*, 2017.

[324] Yiwen Zhang, Yue Tan, Brent Stephens, and Mosharaf Chowdhury. Justitia: Software multitenancy in hardware kernel-bypass networks. In *USENIX NSDI*, 2022.

[325] Zhe Zhang, Zmey Deshpande, Xiasong Ma, Eno Thereska, and Dushyanth Narayanan. Does erasure coding have a role to play in my data center? Technical Report May, Microsoft Research Technical Report MSR-TR-2010-52, May 2010, 2010.

[326] Whitney Zhao and Jia Ning. Project Tioga Pass Rev 0.30 : Facebook Server Intel Motherboard V4.0 Spec. https://www.opencompute.org/documents/facebook-server-intel-motherboard-v40-spec.

[327] Liang Zheng, Carlee Joe-Wong, Chee Wei Tan, Mung Chiang, and Xinyu Wang. How to bid the cloud. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 71–84. ACM, 2015.

[328] Qing Zheng, Kai Ren, Garth Gibson, Bradley Settlemyer, and Gary Grider. DeltaFS: exascale file systems scale better without dedicated servers. In *PDSW*, 2015.

[329] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. Dynamic tracking of page miss ratio curve for memory management. In *ASPLOS*, 2004.

[330] Huaiyu Zhu, Yong Chen, and Xian-He Sun. Timing local streams: Improving timeliness in data prefetching. In *ICS*, 2010.