UNIVERSITY OF CALIFORNIA

Los Angeles

Semantics-Guided Systems Foundations for Disaggregated Datacenters

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Haoran Ma

2024

ABSTRACT OF THE DISSERTATION

Semantics-Guided Systems Foundations for Disaggregated Datacenters

by

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

Resource disaggregation has emerged as a promising solution to enhance both resource utilization and management efficiency in datacenters. Existing disaggregation solutions have largely centered on generic, low-level system optimizations such as minimizing remote access latency at the operating system and hardware levels. However, these solutions often yield suboptimal performance due to the lack of alignment between application semantics and the underlying system layers.

This dissertation presents a novel approach that enhances the performance of disaggregated systems by incorporating application semantics, including memory access patterns, data object ownership, and computational intensity, into the system design. Our methodology is demonstrated through three techniques—Mako, MemLiner, and DRust. Each technique applies program semantics at different levels of the system stack, ranging from programming languages and compilers to runtime environments and operating systems. Specifically, Mako and MemLiner utilize program semantics to develop a new runtime that is optimized for disaggregated memory architectures. Meanwhile, DRust employs data object ownership semantics

in applications to build a programming framework tailored for compute disaggregation.

Collectively, these proposed techniques aim to enhance the performance, efficiency, and consistency of disaggregated systems, making them more viable for practical implementation in today's datacenters. This body of work lays a foundational framework for the co-design and co-optimization of techniques across system layers, aimed at advancing future disaggregated datacenters.

The dissertation of Haoran Ma is approved.

Shan Lu

Todd D. Millstein

Miryung Kim, Committee Co-Chair

Harry Guoqing Xu, Committee Co-Chair

University of California, Los Angeles

2024

Dedicated to my family for their love and support

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to start by giving my deepest thanks to my advisors, Harry Xu and Miryung Kim. Their guidance has been invaluable, and I am deeply appreciative of their support throughout my journey.

I would also like to thank my committee members: Shan Lu and Todd Millstein. Thanks for your valuable feedback on my work, which has been a tremendous help for improving and completing this dissertation.

I am very lucky to have been surrounded by such amazing people in the UCLA SOLAR lab: Yifan Qiao, Chenxi Wang, Shi Liu, Jiyuan Wang, Shan Yu, Jonathan Eyolfson, John Thorpe, Christin Navasca, Arthi Padmanabhan, Pengzhan Zhao, Usama Hameed, Yaoxuan Wu, Zhenting Zhu, Qian Zhang, Jason Teoh, Ben Limpanukorn, Eric Zhou, Fabrice Harel-Canada, Burak Yetistiren, Hongjin Kang, Shu Anzai, Yuanqi Li, Shen Teng, Gaohong Liu, Ricky Fok. Thank you all for the countless enriching research discussions, the enjoyable gatherings, the lasting friendships, and the empathetic support we shared on this academic path.

Lastly, my heartfelt thanks go to my family for their unwavering support as I pursued my PhD. Special thanks to my parents for encouraging me to pursue my dreams and passions. I am eternally grateful for their presence through all my highs and lows.

VITA

Ph.D. Candidate in Computer Science 2019 - 2024
University of California, Los Angeles

B.S. in Computer Science and Technology 2015 - 2019
Tsinghua University

PUBLICATIONS

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

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

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

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

Haoran Ma, Shi Liu, Chenxi Wang, Yifan Qiao, Michael D Bond, Stephen M Blackburn, Miryung Kim, Guoqing Harry Xu, "Mako: a low-pause, high-throughput evacuating collector

for memory-disaggregated datacenters", PLDI, 2022

Chenxi Wang, **Haoran Ma**, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D Bond, Ravi Netravali, Miryung Kim, Guoqing Harry Xu, "Semeru: A Memory-Disaggregated Managed Runtime", OSDI, 2020

CHAPTER 1

Introduction

In recent years, datacenters have emerged as the critical infrastructure underpinning a myriad of digital services—from online shopping and financial transactions to social networking. However, modern datacenters face significant challenges in resource utilization and operational efficiency. Reports from industry leaders such as Google and Alibaba reveal that key resources like CPUs and memory are often underutilized in datacenters, with utilization rates ranging from 40% to 60% [33, 50, 62, 113, 124]. This underutilization results in considerable financial and energy waste.

The fundamental cause of poor resource utilization can be traced to the prevailing server-centric architecture of datacenters. For a long time, datacenters have been organized around the monolithic server as the fundamental unit of deployment, operation, and failure. Each server integrates all necessary hardware resources to execute user programs, with resources being tightly coupled and incapable of scaling independently. This configuration requires applications with diverse resource demands to conform to fixed server setups, often leading to inefficiencies where some resources are fully utilized on a machine while others remain underutilized. Furthermore, the server-centric model presents other significant drawbacks: it lacks hardware elasticity, making it challenging to change hardware configurations post-deployment. It also suffers from a coarse failure domain, as a single faulty component can compromise an entire server. Overall, the monolithic server model severely restricts the efficiency and flexibility of datacenter resource management.

To address these challenges, resource disaggregation has emerged as a promising solution

by redesigning hardware infrastructure [12, 13, 24, 39, 58, 67, 69, 69, 76, 97, 111, 116], networking [17, 34, 44, 49, 56, 63, 82, 94, 108, 115, 123], and software systems [14, 51, 105, 113, 125]. This approach separates resources such as CPUs, memory, and storage from traditional monolithic servers into distinct pools that can scale independently, interconnected by high-speed networks [39, 82] to ensure efficient intercommunication. The benefits of resource disaggregation are threefold: (1) *improved resource utilization*: decoupling resources allows for dynamic allocation tailored to specific application requirements, enhancing overall efficiency; (2) *enhanced failure isolation*: any server failure only reduces the amount of resources of a particular type, without affecting the availability of other types of resources; and (3) *increased elasticity*: hardware-dedicated servers make it easy to adopt and add new hardware.

1.1 Challenges

Despite the potential advantages in resource disaggregation, the practical implementation of it in real-world datacenters faces the following two primary challenges.

Performance Challenge in Memory Disaggregation. In a resource-disaggregated cluster, a single process may span multiple servers. Computations are carried out on compute servers, each equipped with a small amount of local memory. These servers rely on memory servers, which provide extensive memory resources acting as the main memory, while the small local memory serves as a cache. When the memory demand of a process surpasses the local memory capacity, the excess data will be swapped to a remote memory server via the operating system's swapping mechanism. Subsequent access to this data involves notable latency due to data retrieval from the remote server. Despite advancements in architecture [13, 24, 25, 69] and networking technologies [34, 49, 63, 82, 94, 108, 115], as well as OS kernel optimizations [14, 101, 127], a significant latency discrepancy remains between local and remote memory accesses. This difference severely affects the performance of common

datacenter applications like Spark [135], Cassandra [8], and Neo4j [9] when running on existing memory-disaggregated systems. For example, Spark's Transitive Closure experiences a ten-fold slowdown on a swap-based disaggregated system [135], which is impractical for datacenter operations. Enhancing the performance of memory-disaggregated systems is thus critical for their practicality.

Consistency Challenge in Compute Disaggregation. Compute disaggregation allows applications to utilize multiple compute servers, thereby harnessing greater computational power. This distribution of application threads across servers necessitates a shared memory abstraction known as Distributed Shared Memory (DSM). DSM enables multiple servers to share a memory space that appears as shared unified physical memory, though it is physically distributed across multiple machines. However, DSM systems are known to struggle with memory coherence issues. Existing solutions to these issues rely on software-based network communications that emulate single-server hardware operations [27, 31, 32, 46, 74, 89, 128]. These methods involve extensive server-to-server communication, often utilizing RDMA verbs, which incur significantly higher latency than intra-server operations. This increased latency results in substantial overhead and huge performance degradation in applications running on these systems. Therefore, devising an efficient strategy to maintain consistency in compute-disaggregated systems remains a critical challenge.

1.2 Key Insights

The primary limitation of existing approaches to resource disaggregation lies in their predominant focus on the low-level aspects of the system stack, often overlooking the run-time semantics of programs. This oversight results in missed opportunities for optimization, and hence suboptimal performance on these systems. Current memory disaggregation solutions, for example, focus on generic optimizations, such as reducing remote access latency at the OS and hardware levels. Similarly, existing DSM systems typically aim to emulate single-machine

hardware mechanisms to maintain memory coherence, ignoring the characteristics of application behaviors. However, applications inherently offer a wealth of semantic information—such as compute intensity [77, 125], access patterns [105, 126], and data ownership [4]—that can be automatically harvested and leveraged for more informed system decisions.

This dissertation lays the groundwork for integrating a comprehensive range of program semantics into resource disaggregation, developing relevant language constructs, compilers, and system support. Our approach to semantics extraction and integration spans multiple layers of the computing stack. Specifically, this dissertation introduces three innovative resource disaggregation techniques, each utilizing semantic information within workloads to improve efficiency and consistency. The first two techniques exploit compute-intensity and access pattern semantics to optimize the program runtime, addressing performance bottlenecks in memory-disaggregated systems. The third technique utilizes ownership semantics to create an ultra-efficient and consistent compute-disaggregated system. The subsequent sections provide detailed overviews of these three projects, detailing how they leverage semantics to enhance performance and consistency in resource-disaggregated systems.

1.3 Semantics-Guided Disaggregated Runtime

As previously mentioned in §1.1, cloud applications running on memory-disaggregated clusters often encounter significant performance slowdowns. Research studies, such as [77, 105, 125], have shown these slowdowns are primarily due to excessive remote memory accesses. Our investigation into the behavior of these datacenter applications has identified that these accesses predominantly stems from the managed runtime, specifically the garbage collection (GC) mechanisms. Datacenter workloads like Spark [135], Cassandra [8], and Neo4J [9] are largely developed in high-level languages that rely on managed runtimes for memory management. These runtimes automatically manage heap memory through garbage collection, sparing developers from manual memory management complexities. However, our experiments

show that GC activities can amplify the number of remote memory accesses by more than threefold, significantly affecting application performance. The excessive remote memory accesses during GC can be attributed to two primary issues. Firstly, GC operations involve reachability analysis of the whole heap to identify and reclaim unused objects, a process akin to graph traversal, which inherently exhibits poor spatial and temporal locality. Secondly, modern GC algorithms such as G1 [42], Shenandoah [47], and ZGC [2] in OpenJDK are designed to execute concurrently with application processes. This concurrent operation, combined with limited local memory, often results in competition for memory resources. For example, the working sets of GC and applications are often disjoint. When the application needs space, it has to evict data used by the GC, resulting in significant interference.

To address these challenges, this dissertation introduces two novel approaches that harness the semantics of GC to design a runtime optimized for memory-disaggregated environments: Mako and MemLiner. These approaches are elaborated in Chapters 3 and 4, respectively.

Chapter 3: Offloading Garbage Collection to Memory Servers. Our analysis of the managed runtime revealed critical semantics: garbage collection (GC) activities are characterized by low computational intensity but high memory demands. Based on this insight, we propose offloading most GC tasks—such as object tracing, evacuation, and memory reclamation—to memory servers, where the majority of data resides. This approach minimizes interference between application processing and GC tasks and, by moving GC operations closer to the data on remote memory servers, significantly enhances GC efficiency.

Chapter 4: Lining up Garbage Collection and Applications. While offloading GC significantly enhances performance, it introduces deployment complexities within datacenters. Traditionally, GC operations and application operations are seen as independent. However, we have made two key observations that challenge this view. Firstly, objects accessed by the application and those traced during GC are just temporally unaligned. The live objects traced by the GC are mostly accessed by the application at some point during the execution;

the objects accessed by the application must be live objects at the moment of the access and hence they are also the target of GC. Secondly, although changing object-access order in application threads would break the application semantics and hence impossible, altering the object access order in GC tasks, which aim to trace and mark all reachable objects, is possible because the tracing order does not impact the effectiveness of GC. Guided by these observations, we propose MemLiner, a novel runtime memory management technique that lines up the object accesses from GC threads with the application's access patterns. This alignment reduces resource contention by ensuring that the sets of objects accessed by GC and applications during the same period are closely aligned, leading to fewer remote memory accesses and enhanced performance.

These tailored strategies demonstrate how a deeper understanding of GC semantics can lead to significant performance enhancements in memory-disaggregated systems, addressing performance degradation challenge posed by traditional GC approaches.

1.4 Semantics-Guided Disaggregated Programming Framework

The challenge of maintaining consistency in DSM systems has been studied over the years, as evidenced by numerous seminal works [21, 27, 30–32, 46, 54, 68, 74, 75, 83, 91–93, 118, 128]. Traditional methods to ensure memory coherence in these systems typically rely on a critical invariant: for each data block to be accessed, the block is either located on a single node with potential read and write access, or it is replicated across multiple nodes with each having read access only. Prior to a server attempting to access a block, the system must perform multiple synchronization operations across servers to check the state of the block, invalidates copies of that block on all other servers, and then transmits the block to the requesting server. This synchronization process necessitates multiple network round trips. Even with RDMA, the incurred latency is still orders of magnitude higher compared to a single local access, significantly degrading overall performance. Thus, effectively reducing the number

of synchronizations is crucial for minimizing DSM overhead and rendering it feasible for real-world deployment.

Our research has identified a significant opportunity in leveraging semantic information from applications, which is frequently neglected in current systems. Notably, many concurrent programs employ a Single-Writer, Multiple-Reader (SWMR) model to ensure correctness during concurrent operation. Leveraging such information can potentially eliminate the need to check the state of remote data blocks before accessing them, leading to dramatically improved performance. However, a major challenge is how to expose such semantics in a sensible way so that the DSM system can see and act upon it.

Chapter 5: Language-Guided Distributed Shared Memory with Ultra Efficiency. The ownership model [4], which has already been integrated into programming languages like Rust [109], provides an ideal mechanism to convey the SWMR semantics to the DSM system. Rust's ownership type inherently upholds SWMR properties in any compiled Rust program. Leveraging these inherent SWMR semantics, DRust introduces a lightweight, ownership-based coherence protocol, significantly enhancing system efficiency. Building on this protocol, DRust provides Rust-based programming abstractions for DSM along with a runtime that efficiently manages distributed physical resources. This framework enables single-machine Rust programs to operate seamlessly in a distributed environment.

The structure of this dissertation is outlined as follows. Chapter 2 introduces several related basic concepts. In Chapters 3 and 4, we delve into two innovative runtime techniques optimized for disaggregated memory environments: Mako and MemLiner. Chapter 5 presents DRust, a distributed shared memory system with exceptional efficiency and consistency. Finally, we conclude and discuss future directions in Chapter 6.

CHAPTER 2

Background

This chapter offers a comprehensive overview of garbage collection (GC) techniques in managed runtime environments and the ownership model utilized in Rust programming. These foundational concepts are vital for a thorough understanding of the techniques proposed in this dissertation.

2.1 Garbage Collection

In managed runtime environments like the Java Virtual Machine (JVM), the Common Language Runtime (CLR) for .NET, and the PyPy interpreter for Python, automatic memory management is employed through garbage collection (GC). The runtime automatically handles the allocation and deallocation of memory for objects, significantly easing the developers' burden of manual memory management. By doing so, it reduces the likelihood of memory leaks and other memory-related issues. The core principle of garbage collection, as detailed in key literature [65], involves a reachability analysis. This analysis identifies a transitive closure of live objects, allowing the system to reclaim the memory occupied by objects outside this closure. Essentially, modern garbage collection algorithms consist of two primary components: (1) *tracing* the heap graph to identify live objects within that closure, and (2) *reclamation* of memory from dead objects through the evacuation of live objects to contiguous spaces and updating of pointers.

Concurrent GC. To ensure the correctness of pointer updating, a conservative way of running GC is to pause all application threads (*i.e.*, a stop-the-world phase) for full-heap tracing and reclamation. While ensuring pointer updating accuracy, this method introduced significant delays, as reported in [79, 90]. To address this performance limitation, starting from the G1 GC [41], which is the default GC in Oracle's JVM, all modern garbage collectors, including Shenandoah [47] from Red Hat and ZGC [2] from Oracle, run the tracing phase concurrently with application threads to (1) leverage the many available cores and (2) minimize GC pauses. For example, in G1, the number of tracing threads is configured, by default, to be 1/4 of the number of cores.

Although G1 runs tracing concurrently with applications, it evacuates objects in stop-the-world pauses, which can lead to long pause time for large heaps and hence unacceptable to many cloud applications that must meet millisecond-level quality of service guarantees [47]. Compacting even 10% of such a large heap (*e.g.*, dozens of gigabytes of memory) can exceed these pause time requirements. Meeting this level of service agreement requires a GC algorithm which can compact the heap while application threads are running. Shenandoah GC and ZGC are two widely-used low-pause garbage collectors that perform concurrent tracing and evacuation, allowing them to achieve millisecond-level pause times even for very large heaps.

Next, we introduce how concurrent tracing and evacuation work in contemporary garbage collectors.

Concurrent Tracing Algorithm. Logically, tracing divides objects into three colors: *white*, *black*, and *gray*. The white set is the set of objects that are candidates for reclamation. The black set is the set of objects that can be shown to have no references going to objects in the white set, and to be reachable from the roots. Objects in the black set are not candidates for reclamation. The gray set contains all objects reachable from the roots but yet to be scanned.

Initially, all objects are white. Tracing implements a graph traversal algorithm that gradually changes the color of objects reachable from the roots from white to black. For each reachable object o, tracing marks it black, retrieves all objects referenced directly by o, and adds them into the gray set. Each iteration retrieves an object from the gray set, marks it black, and adds more objects into the gray set. The algorithm repeats until the gray set becomes empty; objects that remain white can be safely reclaimed. In practice, a modern runtime uses a bitmap to mark live objects efficiently.

Concurrent Evacuation Algorithm In modern garbage collectors, the heap is typically divided into a set of regions, each with a fixed size. After tracing finishes, a subset of regions with higher garbage ratios are selected as the *collection set* for object evacuation. GC evacuates all live objects identified during the concurrent tracing phase in the collection set to new regions. Concurrent evacuation is prone to data races. A number of techniques have been proposed to prevent races between the mutator and concurrent object evacuator, including Baker's load barrier [19], the Sapphire collector's blocking methods [59], CHICKEN and CLOVER's lock-free methods [98], and the Compressor collector's virtual memory protection mechanism [70]. Oracle's ZGC [2], RedHat's Shenandoah [47], and Azul's C4 [120] are widely used commercial concurrent moving garbage collectors, which use similar treatments to ensure safety in concurrent evacuation. In these three mainstream GCs, they ensure for each live object in the collection set, it must be evacuated *before* application threads can access it. In other words, application threads must access its new location. There are two cases here. First, a GC evacuation thread moves it and then an application thread accesses the old location of the object. In this case, the evacuation thread leaves a forwarding pointer in the object's original location that points to its new location in the heap. When the old location is visited, the application follows the forwarding pointer to access its new location. Second, an application thread accesses the (old location of the) object before the GC evacuation thread moves it. In this case, the application thread moves the object itself, writes the forwarding pointer, and accesses the new location. These three tasks must be executed atomically. After

all live objects in the collection set have been evacuated, GC performs another concurrent heap traversal to update all references to their old locations to the new locations. This reference updating process is usually performed together in the next concurrent tracing phase. Once reference updating is done, the collection set contains only dead objects and can be bulk-freed.

2.2 Ownership Model

Over the past decades, numerous programming languages have been designed to provide safe memory management and data sharing. At the core of such a design is often a tradeoff between memory abstraction level and management efficiency. The ownership concept, and the Rust programming language built upon, are considered promising solutions that achieve a sweet spot between abstraction and efficiency. This subsection provides an overview of the ownership model

Ownership Type. The ownership model has a long history in pursuit of memory-safe language designs and type systems [18, 20, 40, 64, 86, 122]. It has also inspired many systems for safe and efficient resource management [26, 60, 87, 129]. At a high level, ownership enhances a language's type system in a way that guarantees the memory and thread safety of a program with type checking done at compile time. The ownership model encompasses a range of concepts, among which the most important are *lifetime* and *borrowing*.

An ownership-based type system uses lifetime to control the allocation/deallocation of an object. It enforces that each object must have one and only one owner at a time. This allows the compiler to statically track an object's lifetime via its owner, and immediately deallocate the object once its owner goes out of scope, preventing memory leaks without using garbage collection that can introduce disruptive pauses to program execution.

To access an object, a program can create a reference from its owner, but the reference

must "borrow" the permission from the owner, and "return" it to the owner after the access. Specifically, the type system allows the creation of multiple *immutable references* to an object from its owner for concurrent reads but prohibits any write with these references. It allows only one mutable reference to the object only when no other (mutable or immutable) references exist. Through borrowing, the ownership type disallows simultaneous writers and hence prevents data races. In addition, references must return the borrowed permission when they go out of scope. For any program that demonstrates type soundness, the type checker guarantees that references of an object can only reside within the object's lifetime; the object can be safely and automatically deallocated at the time it loses all its references.

Finally, ownership can be transferred from one owner to another—*e.g.*, at a function call, the creation of a thread, or message passing (*i.e.*, via `channel`). However, the type system enforces that ownership transfer must occur in the absence of "borrowing". In other words, no other references can exist in scope when transferring the ownership, preventing data races during ownership transfer.

The guarantees provided by the ownership model with respect to object lifetime and data sharing can be summarized with the following four invariants:

1. **Singular Owner**: each value has one single owner at any time (which must also belong to one single thread).
2. **Safe Borrowing**: All references are created from the owner; permission borrowing and returning guarantees that references that can be used to access the object must be valid.
3. **Single Writer**: Each object allows one mutable reference at most and it cannot coexist with any other references in the same scope.
4. **Multiple Reader**: Multiple references are permitted only when all of them are immutable.

The last two invariants are commonly called the single-writer-multiple-reader (SWMR) property in the DSM literature [85].

```
1  pub struct Accumulator { pub val: Box<i32>, }
2  impl Accumulator {
3    pub fn add(&mut self, delta: &i32)->i32 {
4      *self.val += *delta;
5      *self.val
6    }
7  }
8  fn main() {
9    // Allocates two integers in the heap.
10   let val: Box<i32> = Box::new(5); // val is an owner.
11   let mut b: Box<i32> = Box::new(0); // b is an owner.
12   // Ownership is transferred from val to a.val
13   let mut a = Accumulator{val};
14   { // Only one mutable reference is allowed.
15     let mutr: &mut i32 = &mut *b;
16     // No other reference is allowed now.
17     /* let another_r = &*b; */ // COMPILE ERROR!
18     *mutr = 10; // b == 10
19   }
20   { // Multiple immutable references are allowed.
21     let (b_r1, b_r2): (&i32, &i32) = (&*b, &*b);
22     // Mutable reference is prohibited now.
23     /* let b_mutr = &mut *b; */ // COMPILE ERROR!
24     // Passing by references won't transfer ownership.
25     let sync_add = a.add(b_r1); // a.val == 15
26     let sync_add = a.add(b_r2); // a.val == 25
27   }
28   {// Ownership of a and b is moved to the new thread.
29     // No reference should or can borrow a or b now.
30     let async_add = thread::spawn(move ||
31       a.add(&*b) // a.val == 35
32     ).join(); // lifetime of a and b ends
33     // Current thread cannot access a and b anymore.
34     /* println!("{}", a.val); */ // COMPILE ERROR!
35   }
36 }
```

Listing 2.1: A simple accumulator implementation in Rust.

Rust Language. Rust offers a practical implementation of ownership and is designed with a range of zero-cost abstractions for efficient fine-grained resource management.

Listing 2.1 exemplifies a simple accumulator implemented in Rust (Lines 1–7). The `Accumulator` struct keeps an integer `val` and exposes an interface `add` to increment the value. Rust uses a smart pointer type `Box<T>` to store values on the heap; this pointer serves as the initial owner of the referenced value, as shown in Line 10 and 11. Line 13 instantiates `Accumulator a`, where the ownership is implicitly transferred from `val` to `a.val` during its initialization. Rust allows the creation of mutable and immutable references to access the value. For example, Lines 14–19 create a singular mutable reference (`&mut`) to `b` and set its value to 10. Similarly, Lines 20–27 create two immutable references (`&`) to `b` and add them to

`a` via two function calls. Note that passing references as arguments in function calls does not transfer their ownership.

Finally, Rust allows spawning new threads for concurrent programming, as shown in Lines 28–35. A new thread is created via `thread::spawn`, where the use of `move` captures `a` and `b` in the current scope and transfers their ownership to the newly spawned thread. Rust performs shallow copying for inter-thread communication, where only the pointers stored in `a` and `b` are transferred to the child thread while the actual values on the heap are not moved. Rust guarantees memory safety of `a` and `b` by tracking their ownership. At Line 32, when the child thread finishes its closure (*i.e.*, not necessarily after `join`), and `a` and `b` exit the scope (to which their ownership belongs); their lifetime terminates and Rust deallocates them from the heap.

CHAPTER 3

Offloading Garbage Collection to Memory Servers

Cloud applications, predominantly developed in high-level languages such as Java that utilize managed runtimes, face significant performance challenges in disaggregated clusters due to the garbage collection (GC) mechanism embedded in these runtimes. The challenges stem from the inherent poor data locality of GC and the competition for resources between GC threads and application threads. This chapter introduces Mako, an innovative concurrent and distributed garbage collection technique tailored specifically for memory-disaggregated systems, aiming to mitigate these challenges effectively.

At the core of Mako's effectiveness lies its critical insight into the semantics of garbage collection (GC) process, which is marked by low computational intensity yet high memory requirements. Leveraging this understanding, Mako introduces a distributed architecture where the GC tracing and evacuation is offloaded to remote-memory servers. This not only minimizes the competition for resources but also places GC tasks near the data location, reducing the impact of poor locality. This approach brings in an average of 3× performance speed up compared with traditional concurrent GC algorithms, as GC operations can now run in parallel with application threads without interference. Additionally, with the tracing and evacuation tasks taking place near the data, GC operations become more efficient. However, this model presents a major challenge: maintaining synchronization across servers without shared memory. To address this challenge, Mako employs a novel solution based on the *heap indirection table* (HIT). The HIT contains entries that facilitate single-hop indirection for heap pointers, streamlining the synchronization process across servers. Detailed discussion on the HIT and its pivotal role is provided in Section 3.3.

3.1 Overview

This technique focuses on an environment where memory is disaggregated [105, 125]—a CPU server runs multiple applications and these applications access data located on multiple memory servers. The CPU server maintains a small amount of local memory, which is used by each program as a *software-managed inclusive cache*.[1] Each memory server has a large amount (*e.g.*, TBs) of RAM but only weak cores (*e.g.*, wimpy ARM cores). The mainstream approach to accessing remote memory [14, 51, 113, 125] is through the paging/swap system in the OS; accessing a virtual address whose physical page is not present in the cache triggers a page fault, which the OS kernel handles by fetching the page data from a memory server via *remote direct memory access* (RDMA). Since each CPU server may run many programs that all share its local memory, the amount of cache space for each program is often small (*e.g.*, <50% of the program's working set).

As a result, spatial/temporal locality is crucial for satisfactory performance. For example, ML training and MapReduce applications that perform streaming accesses over large arrays have good spatial/temporal locality. Thus, because most memory accesses will hit into cache, these programs can still run efficiently, although each actual remote access incurs a nontrivial latency (about 100× longer than a DRAM access). On the other hand, graph analytics applications with little locality suffer dearly from remote access latency, because most accesses will trigger page faults and remote fetching.

Problems. Garbage collection (GC) is such a graph workload without much spatial/temporal locality. Mainstream GC performs *tracing* and *reclamation* to collect dead objects. Tracing traverses a heap graph to identify live objects, while reclamation sweeps dead objects or moves live objects. Both tracing and reclamation are memory intensive without locality. As such, running modern GCs *as is* on the CPU server can slow down a program by 1–2

[1] "Cache" refers to a CPU server's local memory in this chapter.

orders of magnitude [125].

Concurrent GCs collect memory while the mutator runs, providing low pause times. However, in this new memory disaggregation setting, they could suffer more than stop-the-world (STW) collectors from lack of locality. Concurrent GCs often have many GC threads that execute simultaneously with mutator threads. When GC and mutator threads both run on the CPU server (with most data located on memory servers), they compete severely for cache and swap resources (*e.g.*, RDMA bandwidth). For example, the working sets of GC and mutator threads are often disjoint. When the mutator (or GC) needs space, it evicts pages used by the GC (or mutator), resulting in significant interference. Our experiments show that modern concurrent collectors such as Shenandoah [47] can slow down applications by 20×. Although concurrent GCs are necessary for latency-sensitive cloud applications [78], such high overheads are intolerable. Our goal is to develop a **low-pause, high-throughput GC** for latency-sensitive applications running **in a memory-disaggregated datacenter.**

A straightforward idea is to run GC tasks on memory servers where data is located, while running the mutator still on the CPU server. Since GC will be physically separated from the mutator, they no longer compete for resources. In addition, GC can run much faster as it is near data and poor locality is no longer a concern. However, a major challenge with this approach is how to enable the intimate interactions needed between the mutator and GC to guarantee the safety of concurrent memory reclamation. In a distributed setting, it is impossible to port existing concurrent GC algorithms in a straightforward manner. The reason is that there is no efficient way to enforce memory coherence between the CPU and memory servers—an unsolved problem in 30 years of distributed shared memory research. Therefore, even if GC tasks are offloaded to memory servers, existing concurrent algorithms such as Shenandoah/ZGC cannot coordinate these servers due to a lack of efficient fine-grained synchronization mechanisms (*e.g.*, lock and atomic instructions).

Mako. This chapter presents Mako, a distributed and concurrent collector that achieves ~12ms pause times (*i.e.*, 2× lower than Shenandoah) with up to 6× higher throughput on disaggregated memory. Mako does so by offloading both tracing and evacuation onto memory servers, while overcoming the aforementioned synchronization problems with the *heap indirection table* (HIT). The HIT provides one-hop indirection for heap references. In the HIT, each object pointer is represented as the address of an immobile HIT entry, which records the actual address of the referenced object. The HIT is a distributed data structure that consists of a set of *tablets*, each containing entries for objects in a heap region. The HIT can be read/written by both CPU and memory servers: the mutator accesses it on the CPU server upon each object access, while each memory server can access only the tablets that correspond to regions hosted by that server. When a memory server evacuates objects from a region, it only needs to update the region's own tablet to reflect the movement of objects, rather than updating all references to those objects throughout the heap.

The HIT provides three major benefits. (1) It eliminates the need to directly update pointers at both the CPU and memory servers when objects are moved, resulting in a significantly simplified algorithm. (2) It allows for immediate reclamation of an evacuated region rather than relying on another tracing pass to update all pointers to the moved objects. (3) It provides fine-grained synchronization: whenever a memory server evacuates objects in a region, the region's tablet is 'locked' (*i.e.*, invalidated on the CPU server), automatically preventing mutator threads from accessing objects in the region, because accessing objects requires looking up their HIT entries, which have been invalidated.

To reap these benefits, Mako must overcome two challenges. First, using the HIT naïvely would double the number of memory accesses. To reduce indirect accesses, Mako allows stack variables to store direct pointers: whenever a reference is loaded onto the stack, it is converted from the address of a HIT entry to the target object address, using a *load barrier*. Mako uses a short stop-the-world (STW) phase to move objects that are directly stack-reachable to guarantee that these references are appropriately updated, before concurrent evacuation

begins. This optimization effectively reduces the HIT's run-time overhead to only **~15%**, which can be easily offset by the significant savings from offloading GC onto memory servers.

Second, concurrent evacuation on memory servers requires a small pause to determine a set of regions to be evacuated and to evacuate root objects in these regions *during the pause* to ensure stack consistency. A naïve approach to guaranteeing safety is to block mutator accesses to these selected regions after this pause because any access can potentially load a non-root object onto the stack, making the stack inconsistent. However, this approach blocks mutator accesses for the entire span of evacuating all selected regions, which can defeat the purpose of our low-pause design.

Therefore, we develop a novel algorithm that performs evacuation on a *per-region* basis [71]. It does not block mutator access to a region, as long as the region is not being evacuated. To guarantee correctness, when the mutator accesses a region that is in the evacuation set but has not yet been evacuated, we let the mutator evacuate the accessed object *immediately on the CPU server*. This guarantees that any objects accessed before the memory server starts evacuating the region have already been moved to the region's `to-space`. Mako only blocks mutator accesses to the region during its memory-server evacuation. As such, a mutator thread blocks for *at most the time needed to evacuate one single region* (as opposed to all selected regions), which is 5–10ms in our experiments.

Results. We implemented Mako in OpenJDK 13 and Linux 4.11.0-rc8, and evaluated Mako on a range of cloud applications with various cache configurations. Mako achieves a 90th-percentile pause time of **11.98*ms***, which is **2×** lower than that of Shenandoah, and two to three orders of magnitude lower than that of Semeru [125], a G1-based generational GC for disaggregated memory. Furthermore, Mako outperforms Shenandoah in throughput by **2–6×** due to offloading tracing and evacuation onto memory servers. Mako is publicly available at https://github.com/uclasystem/mako.

3.2 Mako Design

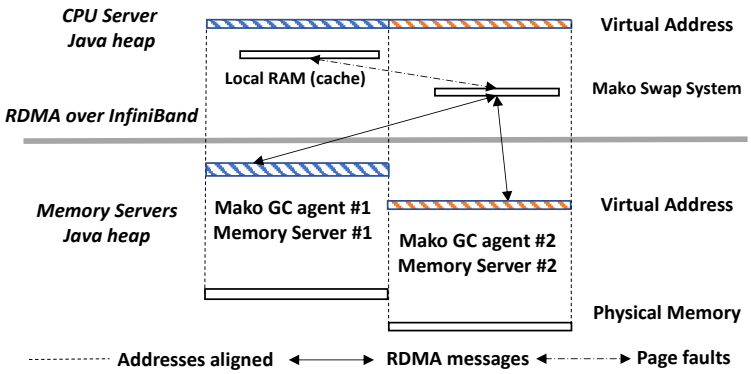


Figure 3.1: Mako's distributed Java heap.

This section provides an overview of Mako's heap structure and distributed GC algorithm.

3.2.1 Heap Structure

Figure 3.1 shows the distributed heap structure we use in our setting. The CPU server runs a JVM with a heap that is logically split into a number of partitions (*i.e.*, address spaces), each backed up by physical memory on a memory server. The CPU server also has a small amount of memory, but this memory serves as a software-managed, inclusive cache and hence is not dedicated to specific virtual addresses. When the mutator accesses pages uncached on the CPU server, a page fault is triggered. Then, the paging system swaps in the pages with needed objects into the CPU server's local memory cache. When the cache is full, selected pages are swapped out to their corresponding memory servers, as determined by their virtual addresses.

Servers are connected by RDMA over InfiniBand. Each memory server runs a Mako agent, which performs concurrent tracing and evacuation over local objects. This agent listens to the CPU server for commands as to what tasks to do and when to do them. Due to its simplicity, the Mako GC agent has a very short initialization time (*e.g.*, milliseconds) and a low memory footprint (*e.g.*, megabytes of memory for metadata). Hence, a memory server

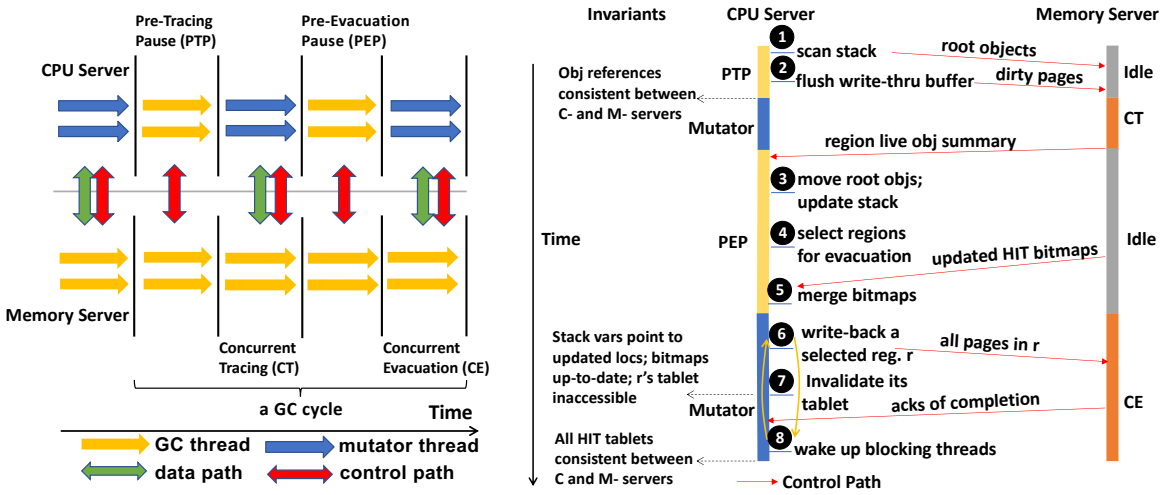
can easily run many agents despite its weak compute (*i.e.*, each for a different CPU-server process). When a Mako agent starts, it aligns the starting address of its local heap with that of its corresponding virtual address range in the global heap maintained by the CPU server. As a result, each object has the same virtual address on the CPU and memory servers, and memory servers can trace their local objects without address translation.

All object allocations occur on the CPU server with regular allocation algorithms. However, if the page on which an object is about to be allocated is uncached, the CPU server's OS will swap the page in from its hosting memory server first before allocation.

Mako uses a *region-based* heap, allowing us to perform concurrent object evacuation at the region granularity. When objects in a region are evacuated on a memory server, the CPU server can still access objects from other regions. Each region has a default size of 16MB, and the CPU server writes back a region if it is selected for evacuation on a memory server. Further details are discussed in §3.4. Like ZGC and Shenandoah, Mako uses one single generation, spanning memory servers. Non-generational collection requires full-heap tracing to identify live objects. However, since tracing and evacuation both occur on memory servers and do not take any compute resources on the CPU server, full-heap tracing has little impact on mutator performance.

3.2.2 Mako's Garbage Collector

Figure 3.2 depicts the high-level design of Mako's concurrent GC. The CPU server runs mutator threads, while memory servers concurrently trace and evacuate live objects they host. As shown in Figure 3.2(a), each GC cycle consists of four phases. Pre-Tracing Pause (PTP) and Pre-Evacuation Pause (PEP) are two short STW phases on the CPU server for synchronization with memory servers, while suspending mutator threads. Concurrent Tracing (CT) and Concurrent Evacuation (CE) are concurrent phases run by each memory server, as the CPU server runs the mutator. Figure 3.2(b) illustrates the main activities in each phase, as elaborated below:



(a) Phase overview (b) GC protocol

Figure 3.2: An overview of Mako's concurrent GC.

Pre-Tracing Pause (PTP). This phase scans the thread stacks (❶), identifies root objects (*i.e.*, reachable directly from stack variables[2]) and notifies memory servers of these objects as tracing roots. Our concurrent tracing builds on the classic snapshot-at-the-beginning (SATB) algorithm [134], which incrementally detects reference overwrites to build the heap snapshot. However, the correctness of SATB depends on an implicit assumption that, at the time tracing begins, all reference updates *made before tracing* are in place; any further updates during tracing will be detected and considered in the heap snapshot. This assumption holds automatically (due to cache coherence) in a single-server setting; however, under memory disaggregation, it no longer holds due to the lack of memory coherence between servers—*e.g.*, a memory server may not see an update made by the CPU server before tracing starts; missing these updates can lead to missing reachable objects in the snapshot.

To solve this problem, PTP must write back all dirty pages to enforce that memory servers see all updates made by the CPU server before concurrent tracing. To minimize this write-back overhead, Mako explores a middle ground between *write-through* and *write-back* by batching page updates in a buffer and flushing the buffer asynchronously when it is full.

[2]For ease of presentation, we focus on stack variables when discussing roots. Our implementation also considers static variables, string constants, JNI references, *etc.* as roots.

When PTP occurs, Mako only needs to flush the pending pages in the buffer (❷).

Concurrent Tracing (CT). This phase starts on each memory server, as soon as PTP finishes on the CPU server. CT performs full-heap tracing. Given that our heap spans multiple servers, memory servers notify each other of cross-server references, whenever they are seen. As a result, each memory server performs graph traversal not only from its own root objects but also from objects with incoming references from other servers. CT finishes when each memory server completes its own tracing and does not have any pending messages from other servers. Mako uses an SATB buffer to record *overwritten values* at pointer updates on the CPU server, while memory servers perform CT. These values are also sent to memory servers and considered as part of the heap snapshot to ensure closure completeness.

Pre-Evacuation Pause (PEP). This phase on the CPU server pauses the mutator to prepare for CE. PEP produces a complete closure by conservatively adding the overwritten values recorded in the SATB buffer into the closure of reachable objects computed by CT. Further, PEP evacuates root objects immediately (❸) and updates their pointers directly *on the CPU server* to guarantee that stack variables all point to updated object locations in the `to-space`. Therefore, concurrent moving involves only non-root objects in the CE phase and does not create any stack inconsistencies. PEP computes a live object ratio for each region—the lower the ratio, the higher the priority—and selects regions for evacuation by CE (❹).

Concurrent Evacuation (CE). When PEP is over, each memory server starts CE to reclaim memory. A challenge here is how to provide synchronization between the CPU and memory servers. As stated earlier in §3.1, the lack of coherence makes it hard to implement fine-grained synchronization primitives. Hence directly applying ZGC or Shenandoah's algorithm would not work in our setting. To overcome this challenge, we use the *heap indirection table* (HIT) to provide one-level indirection for pointer representation in the heap.

Each reference-type object field contains an HIT entry address, whose corresponding value stores the referent's actual address. There is a fixed one-to-one mapping between an HIT entry and a heap object, until the object dies, at which point the entry is reclaimed.

Note that the HIT is conceptually similar to the *object table* design which was used in Smalltalk and in the early days of the HotSpot JVM [3]. However, the HIT is a distributed data structure that manages regions spanning multiple memory servers. The HIT consists of a set of independent *tablets*, each mapping to a region. The CPU server stores the entire HIT metadata but uses the paging system to access specific entries. Each memory server stores the tablets corresponding to their regions. Details of this design can be found in §3.3.

The HIT offers two benefits. First, the HIT significantly simplifies the effort of pointer updating: after an object is moved, Mako only needs to update a single HIT entry, as opposed to updating all of its incoming references (usually via forwarding pointers) in a traditional setting. The HIT helps to guarantee that memory that stores the object can be reclaimed immediately after it is moved. If forwarding pointers were used, memory could not be reclaimed until all incoming references to the object were updated (usually in the next tracing pass).

Second, the HIT provides a fine-grained locking mechanism between the CPU and memory servers during CE. Before evacuating a region, the CPU server writes back all pages in the region (to ensure that the memory server has up-to-date pages; ❻) and *invalidates* the tablet in the HIT corresponding to this region (❼). Write-back is done concurrently so as to avoid a pause. If the mutator accesses an object in a region during its write-back, the mutator moves the object immediately to the `to-space`. After its tablet is invalidated, its hosting memory server will move the rest of the region to its `to-space`. During this process, the mutator cannot access the region due to the lack of valid entries for address translation and hence has to wait in a blocking state. Once the region evacuation is done, the memory server updates the region's HIT tablet with new object addresses and sends an acknowledgment to the CPU server. The CPU server subsequently makes the tablet valid again and wakes up

the blocking threads (❽).

To minimize the mutator's blocking time, CE performs evacuation on a *per-region* basis, repeatedly taking the three steps ❻, ❼, and ❽, until all selected regions have been evacuated. Due to per-region evacuation, the mutator's blocking time is *bounded by the time needed to evacuate one single region*, which is typically small (*e.g.*, $< 5ms$ for 95% of 16MB regions). The evacuation algorithm can be found in §3.4.3.

When basing our GC design on the HIT, to reduce inefficient unnecessary indirection, Mako allows stack variables to point directly to objects instead of using the HIT entries. This is done by using an unconditional load barrier that retrieves the object address from the entry and assigns it to the stack variable, before an HIT reference is loaded onto the stack. Subsequent uses of the stack variable such as calls and field accesses will use the actual object address directly. Conversely, a write barrier is used to convert the object's address into its HIT entry ID before writing the reference to the heap. With this design, the overhead of indirection is incurred only with heap loads and stores of references.

Control vs. Data Path. Mako uses a data and a control path for the CPU and memory servers to communicate. The data path goes through the kernel's normal paging and swap system—pages are evicted based on an LRU algorithm; accessing a page that does not reside in local memory triggers a page fault, and the kernel handles the fault by fetching the page from a memory server. When the mutator executes, it accesses the program through the data path. However, when the GC runs, the CPU server needs to coordinate with memory servers by sending control information, writing back regions, synchronizing the HIT tablets, *etc.* This coordination goes through a control path, implemented via new primitives we add to the kernel.

Pause Summary. Table 3.1 summarizes the three types of pauses introduced by Mako and their time ranges. As shown, PTP and PEP are very short, while the mutator blocking time

during CE is bounded by the time to evacuate one single region, which is also acceptably short.

Table 3.1: Mako's pause time.

Sources of Pause	Type	Time
Pre-Tracing Pause	STW (all threads)	~5ms
Pre-Evacuation Pause	STW (all threads)	~10ms
Per-region evacuation wait	Threads blocking on the region being evacuated	<5ms for 95% of 16MB regions

3.3 The Heap Indirection Table

As discussed earlier, the HIT simplifies pointer updating and provides a fine-grained synchronization mechanism for concurrent evacuation. With the HIT, reference-type object fields no longer store heap addresses and instead store addresses to the HIT entries, each of which stores an actual object address in the heap. A one-to-one mapping is established at allocation between each allocated object and its HIT entry, which remains unchanged throughout its life span.

Tablet. The HIT is a collection of tablets. Each tablet corresponds to a heap region and has three components in Figure 3.3: (1) an array of (word-size) entries, (2) an entry freelist, and (3) a mark bitmap. Each entry in the entry array stores the actual address of an object in the region represented by the tablet. The freelist keeps track of the addresses of free HIT entries, for quick allocation of new objects. The mark bitmap remembers entries whose corresponding objects are marked during tracing. The bitmap is used to construct the freelist for quick entry reclamation. The entire entry array in a tablet is allocated upon the creation of a region; individual entries are assigned to objects upon their allocation.

Distributed Structure. Since each object requires an HIT entry throughout its lifetime, the entire HIT could be too large to fit in the CPU server's local memory. Mako thus stores only the allocation metadata—the tablet's bitmap and freelist—on the CPU server's

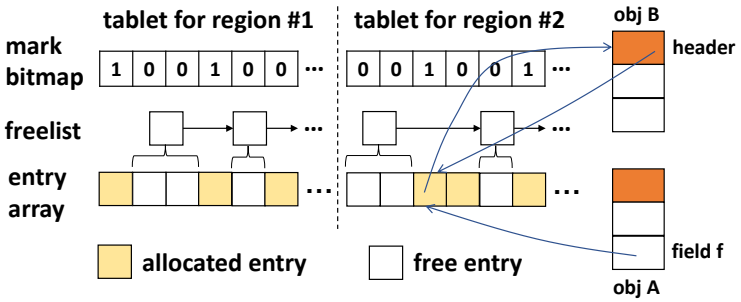


Figure 3.3: The structure of the heap indirection table (HIT). As an example, the field f of object A references object B.

unevictable region but places each entry array on the memory server hosting its region. Entry arrays are subject to paging similar to heap objects.

Since bitmaps are used at both the CPU (in PTP that traces root objects) and memory servers (in CT that traces the full heap), we maintain two copies for each region's bitmap, one on the CPU server and one on the region's memory server. Once CT is done, all live entries are marked. Memory servers send their bitmaps back to the CPU server in PEP, which are then merged to produce the latest liveness information.

Entry Assignment. Upon each object allocation in a region, Mako obtains an HIT entry from the region's entry array by querying its freelist. Mako employs 25 unused bits in an object's header to store the HIT entry ID. Because it uses per-region offsets to represent entry IDs, 25 bits are sufficient. When a direct object reference on the stack is written into the heap, Mako uses this header field to find its HIT entry and write the ID of the entry into the heap (see §3.4.1). When a heap region is created, the entire virtual space for its HIT tablet is allocated, although its backing physical memory is committed incrementally.

The HIT entry assignment, if not done carefully, can be more costly than object allocation. Object allocation can be implemented using an efficient bump pointer algorithm (because evacuation moves objects into contiguous space), but the HIT entries must stay immobile. Hence, to find a reusable entry, Mako must use a freelist.

Since allocation performance is critical to the mutator's throughput, Mako optimizes

the HIT entry assignment by maintaining a *per-thread entry buffer*, similar to the TLAB used in HotSpot [114]. When objects die and their HIT entries return to the freelist, Mako caches a small number of them (*i.e.*, their addresses) in each thread's entry buffer. This optimization provides two main benefits. First, entry assignment can be lock-free as long as this buffer is not empty. Second, entry assignment does not need to go through the freelist when there are cached entries. Furthermore, since entry arrays are located on a memory server, obtaining a free entry (at each object allocation) may need a costly remote fetch. To solve this problem, Mako uses a daemon thread on the CPU server to periodically fill the buffer with new entries and *preload* their pages from memory servers. As a result, the freelist is queried asynchronously and most object allocations can quickly retrieve entries from their thread-local buffers, leading to superior allocation performance.

Reference Resolution. For each object, its reference-type fields now store the HIT entries. Figure 3.3 shows such a representation when `A.f` refers to `B`. There is an additional hop to retrieve `B` from `A.f`. To reduce this indirection-induced latency, we use direct pointers for stack variables so that any method calls or field accesses performed on a stack reference can access the object directly.

Entry Reclamation. After concurrent tracing, Mako begins entry reclamation according to the mark bitmap. Unmarked bits represent entries for dead objects and these entries are returned to the freelist; a subset of them is given to each thread's entry buffers for efficient allocation. Mako performs this step concurrently in a GC thread, when the mutator runs.

3.4 GC Design

3.4.1 Barriers

Algorithm 1: Mako's load/store barriers for reference read/write.

```
1  Function LoadBarrier(a = b.f)
2      HIT entry e ← b.f ;
3      if CE_RUNNING then
4          Region r ← Region(b.f);
5          if r is in the evacuation set s then
6              if IsValid(r.tablet) then
7                  /* r' is to-space; t is the new addr in r' */
8                  t ← Move(b.f, r');
9                  Atomic {
10                     /* only one thread can update *e */
11                     if Region(*e)≠ r' then
12                         *e ← t ;
13                     }
14             else
15                 /* r is being evacuated on a mem server */
16                 while ¬IsValid(r.tablet) do
17                     /* empty loop; wait until tablet becomes valid */
18     /* not in CE or the evacuation of region r is done */
19     a ← *e ;
20 Function StoreBarrier(b.f = a)
21     /* obtain the entry address from a's header */
22     HIT entry e ←Entry(a);
23     b.f ← e ;
```

Heap/Stack Invariant: All stack variables point directly to objects; all heap locations contain the HIT entry addresses.

An important efficiency property Mako maintains is that all stack variables point directly to objects. As such, we use a load barrier (LB) that turns an HIT reference into an object reference upon loading. Conversely, when a reference on the stack is written to the heap, we use a store barrier to retrieve the HIT entry from the object and write the entry address into the heap location. Algorithm 1 shows our barrier logic. Our LB has a fast path that skips all the checks if the execution is not in the concurrent evacuation phase (indicated

by *CE_RUNNING*, which is set by a daemon thread and discussed shortly in Algorithm 2 and 3).

If the execution is in the middle of CE (*i.e.*, Line 3 passes), our LB performs two checks: (1) *evacuation set* check (Line 5) and (2) tablet validity check (Line 6). Our CE algorithm (§3.4.3) selects a set of regions for evacuation and performs evacuation on a *per-region* basis. Hence, if the accessed region r is not in the evacuation set, the mutator follows a fast path that retrieves the address in entry e (Line 19).

If r is in the evacuation set, we perform the second check to test whether the tablet containing entry e is valid (Line 6). $\mathsf{IsValid}(r.tablet)$ returns false if r's tablet is invalidated by the GC thread (Line 5 in Algorithm 3) to prevent mutator threads from accessing r while r is being evacuated by a memory server. At this moment, region r can be in one of two states: waiting or evacuating. First, if r is waiting to be evacuated, Mako still allows mutator threads to access r. r's tablet is still valid and hence the check at Line 6 succeeds.

Before loading $b.f$ onto the stack, the mutator must move the object referenced by $b.f$ to the `to-space` r' of region r (Line 8) and update its HIT entry e with its new address (Line 12). Similar to Shenandoah or ZGC, Mako allows multiple threads to compete when moving the same object in Line 8. However, only one thread can successfully update its entry e to its new location (Line 12); other competing threads, when finding that e has already been updated to point to an address in r' (indicated by $\textsc{Region}(*e)=r'$), give up their object copies and directly use the updated address in $*e$ (Line 19). Here $*e$ denotes the value contained in entry e, which represents the actual object address.

Moving objects upon mutator accesses guarantees that *all objects in r whose references are loaded onto the stack must have been moved to r' on the CPU server before the memory-server evacuation of r starts*. These objects will not be touched by memory servers. Note that we cannot let memory servers evacuate them because their references are already on the stack; if they are still in the `from-space` when the memory-server evacuation runs, moving them makes their stack references stale, creating problems for the mutator.

If the tablet is invalid, region r is being evacuated on a memory server. In this case, we must block the mutator access (Line 16-17); otherwise, the mutator could load a stale reference onto the stack. When r's evacuation by the memory server is done, that server notifies the CPU server, which then makes r's tablet valid again; subsequently, the blocking mutator thread proceeds to execute Line 19.

The logic for the store barrier is much simpler. Both a and b are stack references to objects that must have been moved to the `to-space`, and their HIT entries must have been updated. Hence, writing a's entry address into the object referenced by b will not cause any issue.

3.4.2 Concurrent Tracing

Distributed SATB. The key challenge in Mako's concurrent tracing is the incoherence between the CPU and memory servers, making it hard to implement the SATB algorithm [134] that requires memory servers to see the latest heap references before tracing begins. A naïve approach is to write back all pages cached on the CPU server before tracing during PTP. However, this approach is rather costly as it requires swapping out gigabytes of data while mutator threads are stopped, which can significantly increase the pause time. To solve the problem, we use a variant of *write-through* caching to amortize the swap cost. In particular, we batch page updates during the mutator execution with a write-through buffer—each reference write on a page causes the page to be buffered. When the buffer is full, all pages in the buffer are written back, through the control path, to their hosting memory servers. As the same page may be added multiple times, we deduplicate the buffer before it is flushed. Since this is done asynchronously as the mutator executes, it adds low overhead.

Pre-Tracing Invariant: All object references and their HIT entries on memory servers are up-to-date; memory servers see the latest heap snapshot; the live bits for root objects in the HIT's bitmap are marked.

Due to the use of the write-through buffer, we only need to flush the pending pages in the buffer in PTP, leading to significantly reduced pause time. During PTP, the CPU server scans the stack and sends root objects to their respective memory servers. Before tracing begins, the CPU and memory servers have a consistent view of all heap references. Given that heap locations contain the HIT references, tracing must have access to the latest HIT as well. The HIT entries are handled in the same way as regular data objects—their pages are also subject to our write-through buffering and periodically written back to memory servers.

Mako performs full-heap tracing to compute a complete closure of live objects. To correctly implement the SATB algorithm, the CPU server maintains an SATB buffer. Any pointer updates made by the mutator since the last PTP are captured in the SATB buffer. These updates represent the changes after the heap snapshot is taken. They are sent to memory servers and considered conservatively in CT so that tracing is guaranteed to produce a complete closure that may however include some dead objects [134].

Distributed Completeness Protocol. One challenge in full-heap tracing is how to deal with *cross-server references*—those whose source and target objects are on different memory servers. Tracing in the presence of cross-server references is essentially a distributed graph reachability problem with known solutions [11, 80, 100]. A memory server maintains a *ghost buffer* for each other memory server, which contains messages to be sent to that server. Once tracing hits a cross-server reference, it pushes the target object's HIT entry into the ghost buffer for the object's hosting memory server. Ghost buffers are flushed when they are full. Upon receiving an incoming message, a memory server starts tracing using the object included in the message as an additional root.

However, determining whether all memory servers have completed their tracing work is a challenging task, which requires a distributed protocol. To implement the protocol, we maintain four flags on each memory server:

- `TracingInProgress`: indicating whether the memory server is tracing or idle

- `RootsNotEmpty`: indicating whether the memory server still has pending references received from other servers
- `GhostNotEmpty`: indicating whether this memory server has a non-empty ghost buffer
- `Changed`: indicating whether any of the above three flags changes between the two polls in each cycle

Tracing-Completeness Invariant: For each memory server, all four flags are false.

The CPU server constantly polls those flags on memory servers. In each polling cycle, two rounds of polling are conducted. Upon seeing false values in all four flags on all memory servers in both rounds, the CPU server instructs memory servers to terminate the tracing loop. Note that a memory server does not clear the `GhostNotEmpty` flag until it receives acknowledgments from the receivers, and hence, it is impossible that all flags are false but there are still messages on the go.

The goal of maintaining the last flag (`changed`) and polling twice in each cycle is to avoid the problem of *premature termination*. This problem occurs when the polling of different memory servers happens at different times. For example, memory server **1** receives the poll and tells the CPU server it does not have any work to do while at the same time, memory server **2** is sending references to server **1**. By the time the poll arrives at **2**, these references have already reached **1** and been acknowledged. In this case, server **2** would respond that it is also idle, making the CPU server falsely believe that tracing has finished. We solve this problem using the flag `Changed` on each memory server—for example, if the problem occurs during the first round of polling, the value of `RootsNotEmpty` changes and `Changed` would become true. The second round of polling will detect that and inform the CPU server that tracing is still in progress.

During CT, each memory server marks (its own portion of) the HIT bitmap as live objects are visited. These bitmaps will be sent back to the CPU server at the end of PEP.

Algorithm 2: PEP

```
1  Function PreEvacuationPause
2      /* PEP on the CPU server*/
3      s ← SelectRegionsForEvacuation();
4      foreach Region r ∈ s do
5          r′ ←CreateToSpace(r);
6          /* Evacuate root objects and update all their references*/
7          EvacuateRoots(r, r′);
8      CE_RUNNING ← true; // Set the flag
9      ResumeMutator();
```

Algorithm 3: CE

```
1  Function ConcurrentEvacuation
2      /* GC thread on the CPU server to begin CE*/
3      foreach Region r in s do
4          WriteBack(r);
5          InvalidateAtomic(r.tablet);
6          /* Wait until all mutator threads accessing r leave */
7          WaitForAccessingThreads(r);
8          /* Block mutator's access to r from this point on */
9          Evict(r.tablet.entryarray); // Evict HIT entries of r
10         Evict(r′); // Evict to-space
11         MsgToMemServer("StartEvac", ⟨r, r′⟩);
12         /* Wait here until receiving the ack */
13         while true do
14             if there is a msg ⟨r, r′⟩ from a memory server then
15                 r.tablet.region ← r′;
16                 r′.tablet ← r.tablet;
17                 ValidateAtomic(r.tablet);
18                 Unregister(r);
19                 s ← s \ r; // remove r from evacuation set
20                 if s = ∅ then
21                     CE_RUNNING ← false;
22                 Break;
23     /* Evacuation on each memory server*/
24     Evacuate(r, r′);
25     MsgToCPUServer("Evacuation Done", ⟨r, r′⟩);
```

3.4.3 Concurrent Evacuation

Pre-PEP Invariant: All HIT bitmaps on the CPU and memory servers are consistent and up-to-date.

PEP. When PEP starts, the CPU server sends values recorded in the SATB buffer to memory servers, which use them to finish the final mark. The CPU server combines the HIT bitmaps collected from all memory servers, producing a complete bitmap that reflects the up-to-date liveness information.

Algorithm 2 and 3 describes our algorithm for PEP and CE. During PEP, the CPU server selects regions (Line 3) for object evacuation based upon each region's live object ratio, which is collected during CT by memory servers. The fewer the live objects, the higher priority a region has for evacuation. This is because evacuating objects in regions with more garbage can reclaim more memory.

Once the CPU server determines the evacuation set s, it first evacuates root objects in this set without offloading them to memory servers (Lines 5–7). We update two kinds of references right away in the pause: (1) stack references are direct object references, which are updated to the new locations of the objects; and (2) HIT entries that point to those root objects should also be updated with the new locations. These entries' addresses can be retrieved from the object headers. Note that root objects will not be touched by memory servers after evacuation starts.

One additional constraint here is that objects from the same `from-space` r must be evacuated into the same `to-space` r'. This is because when those objects were allocated, their HIT entries were obtained from the same tablet. Since the HIT entries must stay immobile in the same tablet (otherwise all heap pointers must be updated after evacuation), their corresponding objects must also stay in the same region (although their offsets can change). Finally, PEP sets the *CE_RUNNING* flag (Line 8), notifying mutator threads that concurrent evacuation is starting. This flag will be checked by LB, Algorithm 1.

CE. When PEP finishes, the CPU server resumes the mutator execution. To prepare for CE, the CPU server runs a separate GC thread. For each region r in the evacuation set s, this thread writes back all its pages to its hosting memory server (Line 4). The mutator is allowed to concurrently access objects in r during its write-back: the load barrier in Algorithm 1 at Line 5 will capture the accesses and move the accessed objects to r'. Next, we invalidate r's HIT tablet atomically (Line 5); from this point on, mutator accesses are blocked. At the point of invalidation, there may be mutator threads accessing r. Consequently, we must wait until these threads leave r (Line 7) before letting evacuation begin—Mako invokes `WaitForMutatorThreads` that iterates in an empty loop until no mutator thread is accessing r.

After all mutator threads are blocked, we evict the entire HIT entry array for r (Line 9) and all pages in the `to-space` r' (Line 10). Note that *eviction* is different from *write-back* in that eviction not only writes back the contents of a dirty page to a memory server but also unmaps the page from the CPU server; the next access to the page will have to swap it in from the memory server. We evict r's entry array because the memory server will update these entries during CE and hence those on the CPU server will become stale; eviction essentially forces a "refresh" for its future accesses. Similarly, we evict r' because the memory server will move objects into r' and hence its pages on the CPU server will become stale. After the evictions, this thread sends a command, instructing the memory server to start evacuating r.

> Pre-Memory-Server-Evacuation Invariant: Right before a region r is evacuated on a memory server, objects that remain in r must not have any stack references; none of the pages in $r.tablet.entryarray$ are cached on the CPU server.

Once each memory server receives such commands, it evacuates the remaining objects in the selected regions (from r to r', Line 24). As stated earlier, our treatment guarantees that *objects moved by memory servers must not have any direct references from the stack*. Further, it is impossible for the mutator to turn a non-root object into a root object because mutator accesses have all been blocked during the evacuation. After evacuation is done, memory servers update the HIT entries for the evacuated regions.

Non-root Invariant during CT: Non-root objects that are in the `from-space` r right before r's evacuation remain non-root throughout the evacuation.

The memory server sends a message to the CPU server acknowledging the completion of r's evacuation (Line 25). Upon receiving a message (Line 14), the GC thread on the CPU server unregisters r (Line 18) and makes r' use r's tablet. r is then zeroed out for future allocations. Next, we remove r from the evacuation set s and clear *CE_RUNNING* when s is empty. Mako also validates the tablet for region r (Line 17) so that mutators threads blocking on r can continue (Line 16). We modify the object allocator to *not* allocate into regions in the evacuation set. Hence, allocation will never block on concurrent evacuation.

3.5 Evaluation

To thoroughly evaluate Mako's performance, we selected five cloud applications with large heaps from various sources: H2 (in-memory database), Tradebeans, and Tradesoap (J2EE workloads) from DaCapo [22], and several applications on Cassandra [8] (a NoSQL columnar database) and Spark [135] (a de-facto big data analytics engine), as shown in Table 3.2. These are all widely deployed in industry and represent a wide spectrum of memory-intensive enterprise workloads that dominate the modern cloud. DaCapo programs were executed with huge sizes; For Cassandra, we executed two query workloads (**CII** and **CUI**) with 10 million operations of various types over the popular YCSB [133] dataset. For Spark, we executed PageRank (**SPR**) as well as transitive closure (**STC**) under Hadoop 3.2.1 and Scala 2.12.11.

We compared Mako against two baselines: Shenandoah [47], a modern concurrent collector in OpenJDK, and Semeru [125], a G1-based generational GC for disaggregated memory. Semeru subsumes the vanilla G1 by offloading tracing to memory servers. It was not possible to compare with ZGC [2], another concurrent collector, because ZGC in OpenJDK 13 does not support extending memory to swap partitions, and thus is incompatible with disaggregated memory. In particular, it does not launch when the local memory size is not large enough to

Table 3.2: Systems and applications used to evaluate Mako.

DaCapo [22]	**Size**	
Tradesoap (**DTS**)	DaCapo/huge	
Tradebeans (**DTB**)	DaCapo/huge	
H2 (**DH2**)	DaCapo/huge	
Apache Cassandra [8]	**Operation Composition**	**#Ops**
Insert Intensive (**CII**)	Insert 60%, Update 20%, Read 20%	10M ops
Update & Insert (**CUI**)	Update 60%, Insert 40%	10M ops
Apache Spark [135]	**Dataset & Size**	
PageRank (**SPR**)	Wikipedia Polish [7] (1 GB)	
Transitive Closure (**STC**)	Generated Graph (1.5M edges, 384K vertices)	

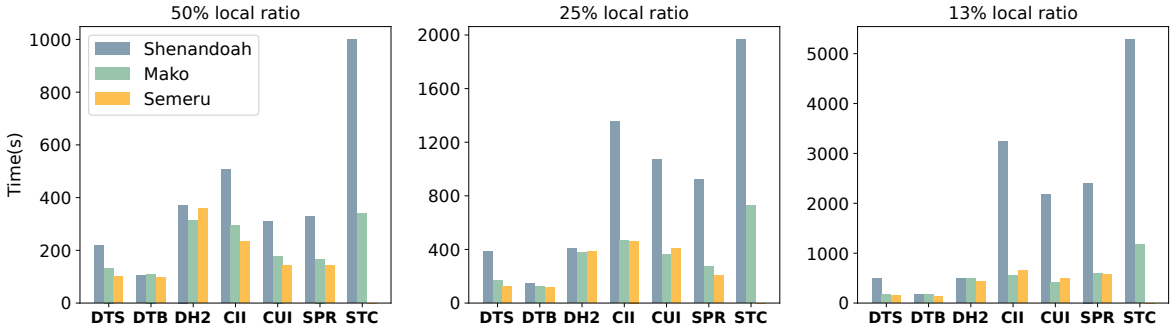


Figure 3.4: End-to-end time under Shenandoah GC [47], Semeru [125] and Mako for 50%, 25% and 13% local memory ratios. Semeru crashed when running **STC** so its bars are not shown.

hold the heap. Further, in the pure local memory setting (*i.e.*, the entire heap is in the CPU server's local memory), ZGC is slower than Shenandoah for 6 out of 7 applications, making Shenandoah a better baseline choice.

We ran our experiments with three machines—one CPU server with two Xeon(R) CPU E5-2640 v3 processors, and two memory servers, each with two Xeon(R) CPU E5620 processors. All of them are equipped with one 40 Gbps Mellanox ConnectX-3 InfiniBand network adapter. They are connected by one Mellanox 100 Gbps InfiniBand switch. One machine runs the JVM process, while the other two machines are used as memory servers. Our experiments used a 32GB heap for Spark and Cassandra and a 16GB heap for DaCapo workloads due to their smaller working sets. Each application was run with three local memory configurations: 50%, 25%, and 13%, representing the percentage of the application's heap that can fit into

the CPU server's local memory. These configurations are enforced with Linux `cgroup` and consistent with the setting used for other memory disaggregation systems [14, 105, 113, 125].

For all applications and all the three configurations, applications used remote memory via swapping. Note that we did *not* follow the conventional way of selecting heap sizes (*i.e.*, multiples of the minimum size that can run the application) because under memory disaggregation, performance of both the mutator and GC depends more on the *local memory size* than the heap size—memory servers can often provide sufficient (remote) memory; hence the heap size is often not a concern. Consequently, we used a fixed heap size for each application but varied local-memory ratios, and ensured that different GCs are compared under the same configurations.

3.5.1 Throughput (End-to-End Performance)

Figure 3.4 reports the end-to-end application time (the lower the better) under Mako, Shenandoah, and Semeru for the three memory configurations. On average, Mako's throughput is **1.75×**, **2.57×**, and **4.10×** higher than Shenandoah under the three ratios.

We observe that the smaller the local memory, the higher the throughput improvement Mako can provide. This is because small local memory implies strong interference between application and GC threads, which compete for local memory and remote memory access bandwidth, leading to severe performance degradation. By moving tracing and evacuation completely off the CPU server, Mako significantly reduces such competition and hence the degradation.

Another important reason for Shenandoah's poor performance is the poor locality of tracing and evacuation; Shenandoah cannot quickly finish a GC cycle on the CPU server, before the heap is full, at which point an expensive full-heap STW GC must run to collect memory. Mako lets both tracing and evacuation run on memory servers, where data is located. Hence, Mako can finish tracing and evacuation quickly and reclaim memory before the heap is full.

Table 3.3: Pause time statistics of Mako (Ma), Shenandoah (Sh), and Semeru (Se) under 25% local memory ratio.

Pause		DTS	DTB	DH2	CII	CUI	SPR	STC
Avg (ms)	Ma	**6.06**	5.54	10.37	**4.63**	**5.34**	**10.13**	**9.90**
	Sh	7.24	**3.67**	**1.40**	8.24	5.48	15.40	26.28
	Se	113.10	345.13	1627.95	1699.54	2463.27	1303.00	701.82
Max (ms)	Ma	**15.34**	**13.78**	21.11	**11.84**	**13.55**	**37.74**	**69.48**
	Sh	86.22	21.03	**8.81**	74.97	118.91	78.21	183.73
	Se	190.52	502.78	3266.01	4323.30	3599.70	5988.406	3066.45
Total (ms)	Ma	**181.78**	183.249	66.99	**333.31**	**272.45**	**658.38**	**1544.71**
	Sh	188.12	**117.27**	**33.61**	1639.21	1614.33	1524.07	5519.67
	Se	2374.96	4486.61	11395.59	79877.97	86214.51	56028.832	N/A

Semeru [125] is a G1-based generational GC that offloads tracing on memory servers, but its STW phase on the CPU server for evacuation is rather long. As shown in Figure 3.4, Mako's throughput is on par with (and slightly lower than) that of Semeru. This is consistent with the community's understanding that concurrent collectors achieve lower pause at the cost of reduced throughput (due to the use of an expensive load barrier, lack of STW phases that can move related objects together to improve locality, *etc.*). To be discussed in §3.5.2, Mako's pause time is up to 1000× lower than Semeru's.

For certain applications such as **CUI** (for the 25% and 13% configurations), Mako achieves higher throughput than Semeru, because Semeru triggers full-heap collections. Semeru performs continuous region-based tracing on memory servers by recording inter-region references into a per-region remembered set. However, these remembered sets quickly grow and contain many stale references, leading to large inefficiencies. In these cases, Semeru's nursery collections cannot reclaim enough memory, and hence expensive full-heap GC is triggered.

Finally, the larger the working set, the more improvement Mako can provide. Mako's improvement is more significant on Spark and Cassandra than DaCapo, because DaCapo applications have a relatively small set of live objects throughout the execution. As such, Shenandoah can run both tracing and evacuation efficiently on the CPU server.

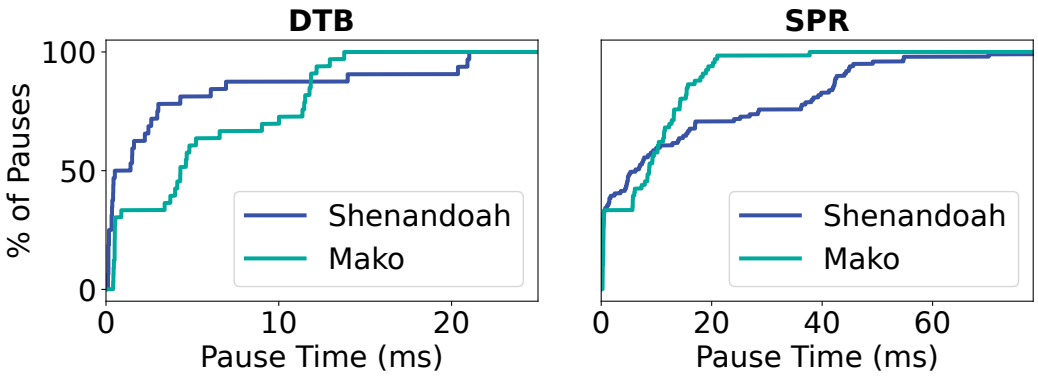


Figure 3.5: Pause time CDF for **DTB** and **SPR**.

3.5.2 GC Latency

This section compares Mako's pause time with Shenandoah and Semeru. Table 3.3 reports the average and total pause times of Mako, Shenandoah, and Semeru for all seven workloads under the 25% local memory ratio. As shown, Mako and Shenandoah's pause times are comparable and both at the level of milliseconds, while Semeru's pauses can be orders of magnitude longer. Again, Semeru crashed on **STC**, so we have no total pause time to report (N/A); for its average and max pause time, we report the statistics before crashing.

To have a close examination of Mako and Shenandoah's pauses, we measure the cumulative distributions of their pause times for the 25% local memory ratio on **DTB** and **SPR** (Figure 3.5). Similar results are observed for the other programs and configurations; these results are omitted due to space constraints.

Shenandoah has more short pauses than Mako due to Mako's synchronizations between the CPU and memory servers, which are not needed for Shenandoah. However, Mako's pause times are much more stable than those of Shenandoah—as shown, the 90th-percentile pause times for Mako for the two applications are $\mathbf{11}\boldsymbol{ms}$ and $\mathbf{18}\boldsymbol{ms}$ vs. Shenandoah's $\mathbf{14}\boldsymbol{ms}$ and $\mathbf{42}\boldsymbol{ms}$ respectively. This is because during tracing and evacuation, Shenandoah touches many uncached pages, triggering page faults and swaps. On the contrary, Mako's tracing and evacuation run on memory servers and have much shorter access time.

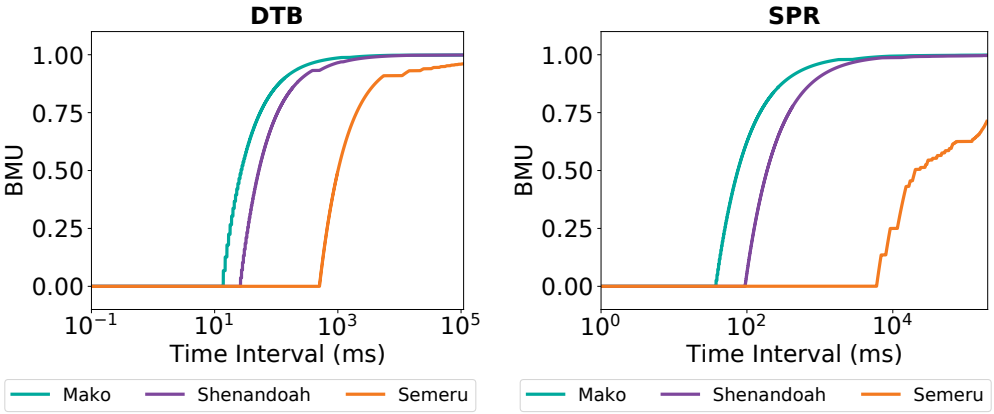


Figure 3.6: Bounded minimum mutator utilization.

To better understand the distribution of collection pauses, we additionally report the *bounded minimum mutator utilization* (BMU) for **DTB** and **SPR** in Figure 3.6. The minimum mutator utilization (MMU) was defined by Cheng and Blelloch [36] as the minimum fraction of the mutator's execution time within any window of a specified size. Sachindran *et al.* [110] extended the definition of MMU to BMU. The BMU for a given window size is the minimum mutator utilization for all windows of that size or greater. BMU measures the fraction of the mutator's execution time over the total run time. For example, if the garbage collector divides a long pause into many short pauses, the impact of these short pauses cannot be captured by just measuring the maximum pause time—we need BMU to understand this impact.

Figure 3.6 depicts the BMU for **DTB** and **SPR**. The X-axis represents different window sizes and the Y-axis shows the percentage of the time spent on the mutator for a given size. For example, the starting point of each curve corresponds to the maximum pause time (*i.e.*, the BMU for any window of a size smaller than this time is 0). As shown, Mako and Shenandoah have similar BMU curves; neither of them has many pauses in a given window (otherwise, the curves would have been much flatter). The BMUs of both Mako and Shenandoah are much higher than those of Semeru due to reduced latency although Semeru outperforms both of them in throughput.

3.5.3 HIT Overhead

This section measures the HIT-incurred overheads.

Load Barrier Overhead. First, the HIT incurs time overhead for address translation on each reference load. It is hard to measure this time directly because (1) load barrier has to run for Mako to work (*i.e.*, there is no way to turn it on and off) and (2) multiple threads run barrier code in parallel, making it impossible to isolate the overhead incurred by one-hop indirection. To overcome these challenges, we ran an emulation: we add the same address-translation logic into an unmodified JVM running Shenandoah, and compared the end-to-end performance between the modified and unmodified JVM. Given that Mako and Shenandoah use the same load barrier, performance differences between these two versions should capture the overhead incurred by indirection.

Table 3.4 reports the additional overhead incurred by Mako's load barrier logic on top of Shenandoah's load barrier. This overhead varies with programs. It is particularly large for **DTB** and **DH2**, where heap reference loads take a significant fraction of the executed instructions. Despite the overhead, running tracing and evacuation on memory servers significantly reduces the mutator-GC interference, improving the performance of both the mutator and GC. As shown in §3.5.1, these improvements are much larger than the barrier-incurred overheads.

Table 3.4: Address translation time overhead.

DTS	DTB	DH2	CII	CUI	SPR	STC
9.41%	16.19%	21.73%	9.69%	6.18%	7.23%	8.81%

HIT Entry Allocation Overhead. The second source of overhead comes from the time needed to find and set up an HIT entry at each object allocation. We used the same emulation-based approach (*i.e.*, using a modified allocator from the unmodified JVM) to measure this overhead. As shown in Table 3.5, for most programs, the entry allocation overhead is much

smaller than the address translation overhead, because object allocations are less frequent than heap reference reads. Mako's thread-local entry buffer usage and preloading reduce this allocation overhead.

Table 3.5: HIT entry allocation time overhead.

DTS	DTB	DH2	CII	CUI	SPR	STC
3.53%	2.41%	1.33%	0.71%	0.83%	1.48%	2.34%

Memory Overhead. Given that each object requires a word-size entry, the HIT incurs memory overhead. Maintaining the HIT's metadata such as freelists and bitmaps requires extra memory. However, the per-object HIT entry pointer in each object's header does not contribute to this overhead, as this header space existed but was unused before. To measure memory overhead, we modified Mako to keep track of all extra memory usage discussed above.

Table 3.6: Memory overhead of Mako.

DTS	DTB	DH2	CII	CUI	SPR	STC
8.64%	14.33%	14.35%	13.62%	14.66%	14.78%	25.61%

As shown in Table 3.6, the overhead varies with workloads and generally falls in the range of 8-15%. The HIT incurs a 25% memory overhead on **STC**, because **STC** must maintain a large number of intermediate results for transitive closure computation, often creating a sea of small objects. The overhead of the per-object entries cannot be easily amortized when the average object size is small. On average, the HIT incurs a 14.7% space overhead, which is often not a concern in a memory-disaggregated datacenter due to a large amount of memory available offered by multiple memory servers.

3.5.4 Collection Effectiveness

The goal of this experiment is to compare the memory reclamation effectiveness of Shenandoah, Semeru, and Mako. Figure 3.7 shows the pre-GC and after-GC memory footprints under the

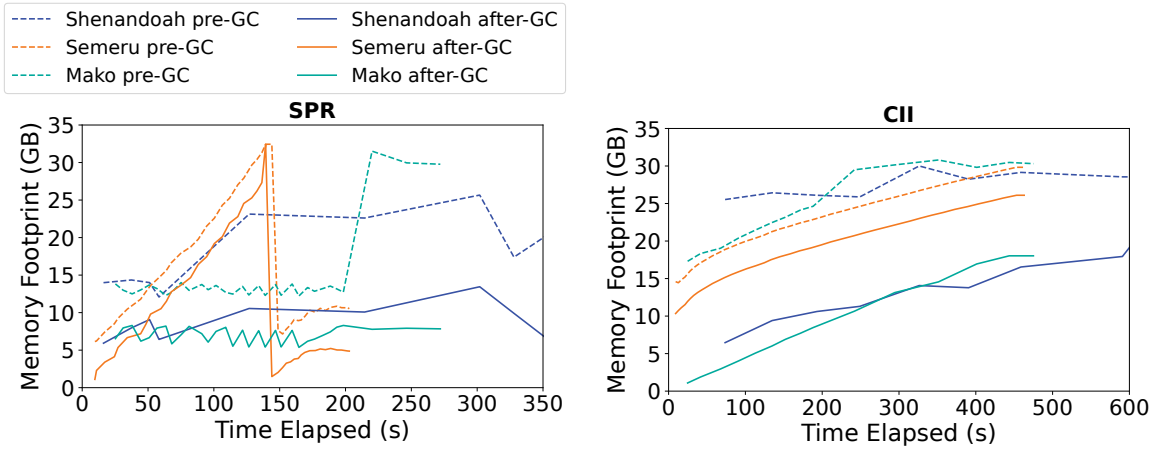


Figure 3.7: GC effectiveness under 25% cache ratio.

25% local memory ratio of **SPR** and **CII** for the first 350 and 600 seconds of the execution, respectively.

As shown, Mako reclaims memory more efficiently than Shenandoah by offloading tracing and evacuation to memory servers. Due to the concurrent and incremental nature of concurrent GCs, the memory footprints under both Mako and Shenandoah are much more stable than those under Semeru.

For **SPR**, Semeru's heap usage keeps increasing as nursery collections run and long-lived objects are continuously promoted to the old generation. Once nursery collections cannot reclaim enough memory, Semeru triggers a full GC and reclaims a significant amount of memory (*i.e.*, the sharp decline of heap usage in Figure 3.7(a)). For **CII**, Semeru does not encounter any full-heap GC; as shown, each nursery collection reclaims a small amount of memory. Mako and Shenandoah can reclaim more memory due to concurrent full-heap tracing and reclamation. Mako finishes much faster than Shenandoah (which actually runs much longer) due to the GC offloading.

3.5.5 Heap Region Size

To understand the impact of the region size, we ran Mako on **SPR** under 25% local memory with two other sizes: 8MB and 32MB. Since evacuation is done on a per-region basis and the pause time depends on the region size, reducing the region size (from 32MB to 8MB) leads to a reduction of the average pause time (from 15.32ms to 8.13ms). However, using a smaller region increases the end-to-end running time (*i.e.*, reduces throughput) by a small margin from 270.99s to 281.59s. This is because a smaller region can lead to higher intra-region fragmentation, resulting in a lower object allocation rate. Figure 3.8 depicts the intra-region fragmentation ratio for **SPR** under three different region sizes, 8MB, 16MB, and 32MB. As shown, the average size of the free space is roughly proportional to the region size.

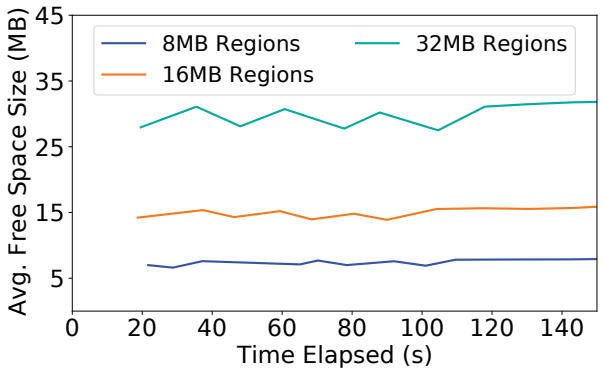


Figure 3.8: The average size of the intra-region contiguous free space for different region sizes.

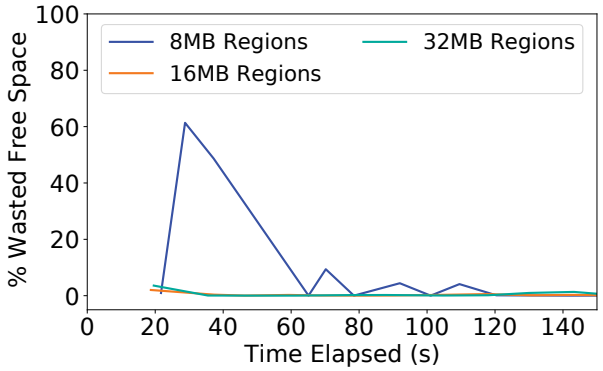


Figure 3.9: Ratio of wasted free space over total heap usage for different region sizes.

Additionally, in OpenJDK, when allocating an object whose size is larger than the free

space of the current region, the allocator simply retires the current region and continues to search for free space whose size is larger than the allocation request in other regions. The free space in the current region is thus wasted. The smaller the region size, the larger the wasted free space.

To quantify this waste, we report the ratio between the sizes of the wasted space and the used heap in Figure 3.9. It is clear that using 8MB regions leads to more space wasted due to severe intra-region fragmentation. These results motivated our choice of using 16MB as the region size, leading to an overall of 10.1ms GC pause time and 272.71s throughput.

3.6 Summary

Mako is the first concurrent evacuating collector that provides low pause times for the emerging datacenter architecture with memory disaggregation. It offloads both tracing and evacuation to memory servers that host the Java heap and leverages the HIT to simplify pointer updating and provide synchronization mechanisms. An evaluation of Mako on a set of modern cloud applications demonstrates that Mako significantly outperforms Shenandoah in both latency and throughput, making it a promising candidate for real-world deployment.

CHAPTER 4

Lining up Garbage Collection and Applications

In Chapter 3, we introduced a GC offloading approach that achieved high throughput and minimal pause times. However, this approach necessitates remote memory equipped with computational resources to execute the GC, an assumption not universally valid in disaggregated clusters. For instance, recent advancements in hardware for memory pooling within disaggregated datacenters, such as CXL-attached memory [39], lack embedded compute capabilities. To overcome this limitation, this chapter presents a novel technique without relying on GC offloading, MemLiner, which aims to line up GC and applications. MemLiner's design relies on two key observations about GC and application semantics. Firstly, the objects accessed by the application and those traced by the GC are not completely unrelated but just temporally unaligned. Most of the live objects traced by the GC are accessed by the application at some point during its execution. The objects accessed by the application must be live at the moment of access and hence the target of GC. Secondly, although altering the object-access order in application threads would disrupt application semantics, changing the object-access order in GC is possible. GC threads focus on tracing and marking all reachable objects in the heap, and the order in which they do so is not critical.

Guided by these semantic insights, MemLiner's core concept is working-set alignment. By reordering the objects traced by GC threads to follow a similar, though not identical, memory-access path as the concurrent application threads, GC and the application's working sets are close to each other. This reduces resource contention, allowing for better application performance. Crucially, MemLiner achieves this without requiring offloading and remains compatible with existing GC algorithms.

4.1 Overview

Most memory-disaggregated systems [14, 51, 81, 116, 125] build on a cache-and-swap mechanism: the application's host server uses local memory as a *data cache*. Once a page that does not reside in the local memory is accessed, a page fault is triggered and the page is fetched from a remote server into the local memory. Good locality and effective remote-memory prefetching [81, 84] are crucial to the performance of applications running in such far-memory systems.

Unfortunately, the interference from garbage collection (GC) severely degrades the memory-access locality and remote-memory prefetching for applications written in high-level languages (e.g., Java, Go, and Python), which are dominant in datacenter workloads. At run time, application threads access heap objects following their program-execution paths, while GC threads *concurrently* scan the heap, performing graph traversal from a set of "roots" (*i.e.*, objects referenced by stack and global variables) to mark live objects. Object accesses by these two sets of threads are uncoordinated, creating two disjoint working sets, as illustrated by Figure 4.1(a), and causing severe performance problems.

Problem 1: Resource Competition. Pages swapped in for GC's heap traversal are often not used (in near future) and hence evicted by the application; conversely, pages swapped in for the application are often not needed (in near future) and evicted by GC. Evicting each other's pages, the application and GC both suffer from severe local-memory misses and further compete for RDMA bandwidth for page swapping. The more concurrent activities a GC runs , the more the resource competition between GC and the application—our results show that running Spark with the Shenandoah concurrent GC [47] on the 25% memory configuration incurs a 12× slowdown to the end-to-end performance, which is 5× larger than the default G1 GC that reclaims memory in stop-the-world pauses.

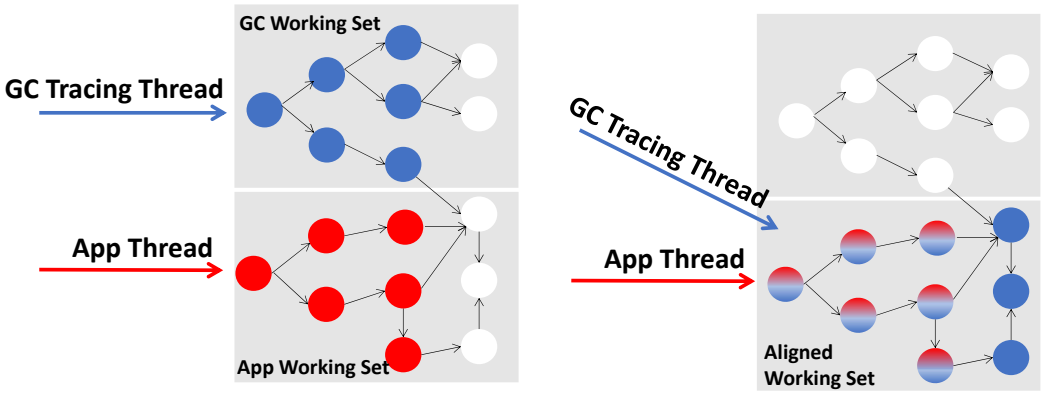


Figure 4.1: Our main idea: the working sets of GC threads, in blue, and application threads, in red, during a time window (a) without or (b) with the access alignment from MemLiner.

Problem 2: Ineffective Prefetching. Monitoring the execution of a managed program, an OS-level prefetcher such as [81] cannot recognize clear memory-access patterns and has to give up prefetching. The reason is that, even if the application's memory accesses follow a simple sequential pattern, the combined accesses from both the application and the GC often appear random from the OS' perspective.

State of the Art. In the past, supporting applications that have large memory footprints (*e.g.*, larger than the main memory size) is not the priority of traditional GC. Although there exists a body of work (such as Platinum [131]) on concurrent GC, such work focuses primarily on improving throughput and reducing latency on memory-abundant servers. However, remote memory is designed to enable applications to use more memory than what their hosts can offer; as a result, developing new GC techniques to support these applications becomes a crucial task.

Recent work Semeru [125] supports running Java programs on disaggregated hardware by disaggregating the traditional JVM into two new ones, with the CPU-JVM executing the program on the CPU server and the memory-JVM performing GC on the memory server. The idea of offloading GC completely to a remote server works for Semeru where *all* the application's memory data is located in a remote server, but does not suit today's

datacenters where resources are not entirely disaggregated and applications use remote memory only if their local memory runs out. Furthermore, this offloading approach imposes extra communication overhead for CPU-JVM and memory-JVM to coordinate, and extra computation cost on the remote memory server to run the memory-JVM, which may impose deployment challenges.

Another recent work AIFM [105] proposes a novel runtime to improve the prefetching and swap performance of applications running in remote-memory systems. AIFM targets applications written in native languages (C/C++), and hence cannot easily be applied to solve the GC interference problem in the managed language runtime.

MemLiner. This chapter presents a fully-automated runtime technique, MemLiner, for programs written in high-level languages (HLLs) to efficiently use remote memory.

The design of MemLiner is based on two key observations.

First, the objects accessed by the application and the GC are not completely unrelated—they are just not temporally aligned. The live objects traced by the GC are mostly accessed by the application at some point during the execution; the objects accessed by the application must be live objects at the moment of the access and hence the target of GC.

Second, although changing object-access order in application threads would break the application semantics, changing that order in GC would not. Specifically, GC threads aim to trace and mark all reachable objects in the heap, while the *order* of that tracing and marking (*e.g.*, which objects are traced first) does not matter.

Guided by these observations, the key idea behind MemLiner is *working set alignment*. MemLiner carefully reorders the objects traced by the GC threads, so that they follow a similar, although not identical, memory-access path of the concurrent application threads (illustrated by Figure 4.1(b)). Consequently, their working sets can better overlap with each other; the resource competition can be much alleviated, with much reduced page faults and on-demand swaps; the application's access patterns can be more easily recognized by the

underlying prefetcher such as Leap [81]. All of these are achieved in a way that is compatible with existing GC algorithms, without offloading the GC to another machine or re-desgining the prefetcher.

MemLiner must overcome several challenges.

First, *how to align GC threads with application threads*. In a conventional setting, GC traces objects using a graph traversal starting at the root objects. To align GC's accesses with application threads', MemLiner uses a *priority-based* algorithm—MemLiner makes application threads inform the GC of the objects they are accessing; these objects, which must be live and reachable in the object graph at that moment, are then immediately traced and marked by the GC, without any risk of triggering page faults and expensive remote swaps. To enable such communication, MemLiner leverages the *read-write barrier*—a piece of code executed by the runtime at each heap read/write in the application—to inform GC of the objects on the application's access path. Details of the coordination are discussed in §4.3.1.

Second, *when to break the alignment* so that GC can finish its work without unnecessary delays. Completely aligning GC threads with application threads could severely delay GC from reclaiming dead heap space, as application threads may take a long time, sometimes even the whole execution, to access every live object. In fact, a complete alignment is unnecessary, as application threads may repeatedly access the same object in a short time window due to application semantics, like during a loop, while GC only needs to mark that object live once. Consequently, MemLiner allows GC to break from the alignment to work on another part of the heap traversal from time to time. To minimize the interference, MemLiner prioritizes two types of objects in GC's unaligned accesses: (1) objects that will likely be accessed by the application soon; (2) objects that were accessed by the application not long ago and hence are likely still inside the local memory. The former is predicted based on what objects the application just accessed; the latter is predicted based on object-access history that MemLiner efficiently encodes inside the per-object pointer. Details can be found in §4.3.2.

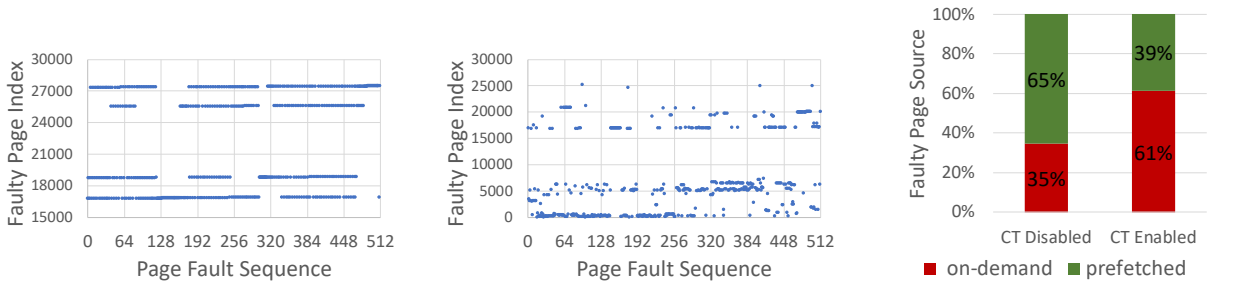
Results. We have integrated MemLiner into two widely used GCs (G1 and Shenandoah) in OpenJDK 12. A thorough evaluation with Spark, Cassandra, Neo4J, QuickCached and DayTrade demonstrates that MemLiner improves the end-to-end execution time by an overall of **1.48**× and **1.51**× under the 25% and 13% local memory configurations for the G1 GC, and **2.16**× and **1.80**× for the Shenandoah GC (which runs concurrent GC threads more frequently than G1). Furthermore, MemLiner improves Leap's prefetching coverage and accuracy by **1.5**× and **1.7**×, respectively. Compared to Semeru [125], MemLiner achieves a comparable performance without offloading any computation on remote servers.

Key Takeaway. Although there are several directions of work on remote memory (*e.g.*, clean-slate approaches such as AIFM [105] and Kona [28], swap optimizations such as InfiniSwap [51] and FastSwap [14], as well as distributed runtimes such as Semeru [125]), MemLiner takes an *easy-to-adopt, non-intrusive* approach that enables performance improvements for a wide variety of new and legacy applications. MemLiner is orthogonal to (and complements) these existing techniques—aligning the memory accesses between application and GC threads reduces thread-level interference and the application's local-memory working set regardless of the underlying remote-access mechanisms and optimizations.

4.2 Motivation

In this section, we use an experiment to quantitatively demonstrate (1) how tracing and application threads interfere with each other, and (2) why simply disabling concurrent tracing cannot solve the problem.

Setup. We ran Spark Logistic Regression (LR) with the Wikipedia dataset on OpenJDK 12 and its default G1 GC. We used two machines, each with 2 Xeon(R) CPU E5-2640 v3 processors, 128GB memory, 1024GB SSD, and CentOS 7.5, connected by RDMA over a 40Gbps InfiniBand network. One machine runs Spark, using local memory and remote



(a) Page fault trace of Spark LR.

(b) Page fault trace of Spark LR with CT.

(c) Prefetching effectiveness.

Figure 4.2: Prefetching effectiveness for Spark LR executed atop OpenJDK 12 (with its default G1 GC): (a) trace of faulty page index for application threads only; (b) trace of faulty page index when concurrent tracing (CT) is enabled; (c) disabling CT significantly improves the effectiveness of Linux' default swap prefetcher.

memory on the other machine. We configured the first machine to have just enough memory to host 25% of Spark's working set. We name the first server providing compute resource as *host server* and the second server providing remote memory as *remote server*.

We compare the execution of Spark LR in two modes:

(1) The G1 GC's concurrent tracing is *disabled*;

(2) The G1 GC's concurrent tracing is *enabled*—the default option in G1 GC. The number of tracing threads is set to be a quarter of the number of available cores, as suggested by G1.

In both cases, the heap size of Spark LR is set to 32GB and the host server can hold up to 8GB of its heap. The execution goes through application-execution phases and stop-the-world GC phases alternatively.

How much interference from concurrent tracing? To have an intuitive look at how well prefetching may or may not work, we randomly sampled 512 *consecutive* page faults in the middle of Spark LR's execution under both execution modes. Note that, since we collected page-fault information from inside the kernel and the execution under the two GC modes proceeds at vastly different paces, we cannot guarantee that the two samples come from the same window of application instructions, but we do make sure that the stop-the-world GCs

did not occur during our samples.

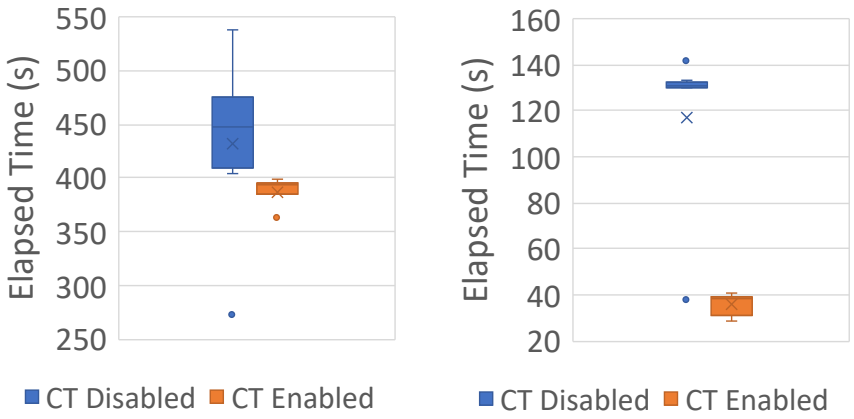
Figure 4.2 (a) and (b) illustrate the virtual page index of the faulty addresses (Y-axis) ordered by when each fault occurs, with the sequence number shown in the X-axis. Without concurrent tracing, each of the application threads has a clear streaming access pattern, as shown in Figure 4.2(a), which should be detected by an advanced prefetcher. This clear pattern is messed up by concurrent tracing, as shown in Figure 4.2(b), making prefetching much harder.

To quantitatively measure the impact of concurrent tracing on prefetching, we checked 500 application-execution phases (*i.e.*, the period between two stop-the-world GCs) to understand, among all the page faults, how many were resolved through on-demand swaps from remote memory and how many were resolved using data already brought in through prefetching. Clearly, this ratio of on-demand swapping versus prefetching directly affects the application performance.

As shown in Figure 4.2(c), without concurrent tracing, prefetching is effective, addressing 65% of the page faults. Unfortunately, with concurrent tracing, this ratio greatly dropped to only 39%, with the remaining 61% of page faults leading to costly remote-memory accesses. Note that our experiments use Linux's default swap prefetcher. If an advanced prefetcher such as Leap [81] is used, the prefetch-ratio would be even higher without concurrent tracing and hence suffer even more from the interference (see §4.6).

Finally, to understand how much the interference has affected the working set of the execution, we also measured the average number of page faults encountered by application threads. The page-fault rate jumps from **3.5K** per second per thread to **9.6K** per second per thread, when concurrent tracing is enabled, indicating a huge interference.

Why not just disable concurrent tracing? Having seen significant interference from concurrent tracing, a strawman solution is to simply disable concurrent tracing for applications running in far-memory systems.



(a) End-to-end execution time. (b) GC pause time.

Figure 4.3: Concurrent tracing improves overall performance. (Data is from 10 runs of each program; dots are outliers.)

Unfortunately, this strawman solution does not work. First, modern concurrent GCs such as Shenandoah [47] and ZGC [2], which are designed for low-pause and used widely by latency-sensitive cloud applications, rely on concurrent tracing to reclaim memory (also concurrently). Disabling concurrent tracing would destroy the functionality of such collectors. Second, even for GCs such as G1 that could perform tracing in a stop-the-world phase, the end-to-end execution time suffers significantly without concurrent tracing. As shown in Figure 4.3(a), the execution time increases by **18%** on average in 10 runs. The main reason is that the aggregated stop-the-world GC periods now take **2.7×** longer without concurrent tracing, as shown in Figure 4.3(b). Without concurrent tracing, each (fast young-generation) GC cannot reclaim as many dead objects in the same amount of time and has to resort to *slow, full-heap GC* that scans and compacts the whole heap space in a stop-the-world period, which is extremely time consuming. For example, the longest full-heap GC (*i.e.*, a single pause) in Spark LR takes 76.9 seconds, clearly an intolerable delay.

Key Takeaway. Memory accesses from application and GC threads exhibit diverse patterns, significantly increasing the application's working set and making prefetching harder. Simply disabling concurrent tracing in GC would not work, as it reduces the number of local-memory

misses at a cost of significantly increased GC pause and end-to-end execution time. MemLiner offers a solution that can greatly reduce the number of local-memory misses and increase the effectiveness of existing prefetchers without introducing extra GC-pause time, and hence effectively reduce the end-to-end execution time.

4.3 MemLiner Design and Implementation

This section presents the design and implementation of MemLiner, particularly how we realize the two key ideas: (1) making GC concurrently trace objects immediately after their access by application threads (§4.3.1) and (2) making GC trace other live objects through a novel priority-based algorithm (§4.3.2) to reduce interference.

MemLiner modifies the garbage collector inside the runtime and the swapping system inside the kernel, while requiring *no* changes to applications. In terms of runtime changes, MemLiner is a general mechanism that can be integrated into any modern runtime that performs concurrent tracing. This technique focuses on a design for Oracle's OpenJDK, a commercial JVM that supports a variety of high-level languages such as Java, Scala, Python, Ruby, *etc.* In terms of kernel changes, we build MemLiner atop paging/swap mechanisms that already exist in the OS kernel, with minimal invasion. Any swap optimizations such as InfiniSwap [51] and FastSwap [14] can be readily used to improve the swap performance for a MemLiner-equipped runtime. MemLiner's runtime design is independent of how remote memory is accessed; for example, MemLiner could also run on a clean-slate platform such as Kona [28] that access remote memory based on cache coherence, not page faults, if coherence is provided by hardware.

When a MemLiner-equipped JVM is launched, the maximum heap size $\boldsymbol{M}$ is specified by the user via a command-line option. A small amount of physical memory on the local machine is initially used to back up the heap (which is much smaller than $\boldsymbol{M}$). The heap stays entirely in local memory until its usage exceeds the size of local memory, in which

case, the OS kernel allocates remote memory by registering it as an RDMA buffer. The kernel uses an approximate LRU algorithm to evict pages. MemLiner does not require any software/hardware support on remote servers, providing a practical solution that can be readily used in today's cloud.

4.3.1 Application and GC Coordination

To align memory accesses, application threads inform GC's tracing threads of the objects they are accessing so that tracing threads can trace these objects immediately.

To facilitate such communication, we need to instrument every heap read/write instruction so that the application can send an object pointer to GC when it dereferences the pointer: *(1)* At a statement that reads an object field or an array element of the form $a = b.f$ or $a = b[i]$, our instrumentation pushes the corresponding address in b into a thread-local producer-consumer queue (PQ), which will be read by GC during tracing. *(2)* At a statement that writes an object field or an array element of the form $b.f = a$ or $b[i] = a$, we similarly push the object reference in b into the PQ.

MemLiner implements this instrumentation through existing read/write barriers—a piece of code that is executed by modern runtimes at each heap read/write operation to record heap information for GC purposes. MemLiner piggybacks on the existing implementation of read/write barrier in OpenJDK 12 that intercepts both interpreted and compiled code. A PQ is created for each application (producer) thread so that no synchronization is needed for enqueuing pointers. A GC tracing (consumer) thread constantly checks PQs to retrieve pointers for tracing. Consumer threads use atomic instructions when dequeuing object pointers. In practice, the number of application threads is often larger than the number of tracing threads; hence, there is little contention when PQs are accessed by multiple threads.

To minimize the maintenance overhead, we represent each PQ as a non-blocking *ring buffer*. Producers and consumers do not synchronize at all—an application thread keeps writing into the queue even if it is full. As such, the application thread may overwrite entries

that have not yet been picked up by GC. Note that this would not cause any correctness issues because those entries only indicate *tracing priority*: overwriting an entry will delay the corresponding object's tracing, but the tracing of these objects will eventually happen in GC's regular graph traversal, which will be discussed in the next sub-section.

Note that our instrumentation code at different program points is unlikely to enqueue the same object reference multiple times (*e.g.*, neighboring reads to the same data structure). This is because marking an object live sets a bit in a global live bitmap. Before pushing each object reference into the queue, an application thread checks its bit from the bitmap and filters it out if the bit is already set.

4.3.2 MemLiner Tracing Algorithm

4.3.2.1 Design Overview

A major challenge in aligning tracing and application threads is that GC has to compute a *full* closure of live objects to reclaim memory. Hence, it is unproductive to trace a live object only right after it is accessed by the application, which will delay the closure computing, leading to inefficiencies in memory reclamation.

The key question here is: *how can GC make quick progress in closure computation without producing a working set that significantly departs from that of the application?* On the one hand, after processing all objects in the PQ, we want GC to trace as many other live objects as possible, even if not in the PQ, to complete the closure. On the other hand, GC should better *not* trace many objects that do not reside in local memory because tracing those objects triggers page faults and swaps. How to reconcile these seemingly conflicting goals is a problem MemLiner must solve.

Reachable Object Classification. To better explain our tracing algorithm, we first classify all live objects at any moment of the execution into three categories based on their

location and when they are accessed by the application, as illustrated in Figure 4.4[1]:

*(1) Objects in local memory (*i.e.*, data cache):* These objects have recently been accessed by the application and have not been evicted yet. Clearly, tracing them at this moment (or in the near future) would not generate any page faults or interfere with the application. Many of these object (*i.e.*, the red ones in the figure) are made known to the GC through the PQ discussed in §4.3.1. However, since the PQ is designed to be a ring buffer, some of these objects (*i.e.*, the striped ones in the figure) may be missed by GC due to being overwritten in the ring buffer. How to trace them sooner rather than later requires extra handling that we will discuss later.

(2) Objects in remote memory and to be used soon: Since these objects (*i.e.*, the wavy nodes in Figure 4.4) will soon be accessed by the application, they are typically just a few references away from the objects being accessed by the application. Tracing them is also desirable—although they are currently not local, they will soon be needed by the application. If GC triggers page faults when accessing them, the costs of handling these faults and swapping would be *necessary* as they are "prepaid" by GC for the application.

(3) Objects in remote memory and not used soon: These are illustrated as clear-circle objects in the figure. They were used by the application a while ago and got evicted to remote memory. Tracing them is needed *eventually* but is undesirable now or in the near future, as tracing them pays the high cost of fault handling and swapping (which is entirely wasted if they are not used by the application before their next eviction).

Handling Different Categories in GC. MemLiner's central design goal is to let GC trace objects in Category (1) and (2) right away *to maximize progress* and delay tracing objects in Category (3) *to avoid unnecessary page faults and interference.* Among the different categories of objects, our starting point is the set of red objects, which are captured by the

[1]For ease of discussion, here we do not consider cold objects staying in cache due to hot objects on the same page. We will discuss it in Section 4.3.3.

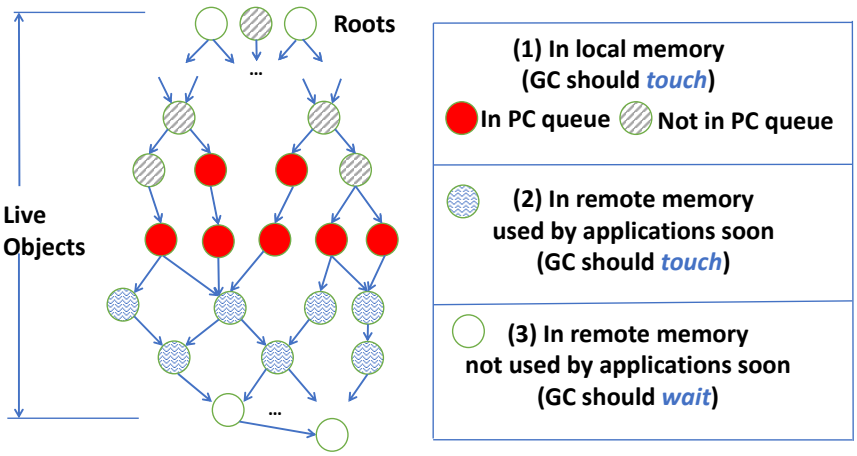


Figure 4.4: Classification of reachable objects in the heap: red objects are being accessed by the application and shaded objects are what MemLiner intends to trace.

read/write barrier, sent to GC via the PQ, and traced by GC immediately.

With the red objects in hand, the wavy objects in Category (2) are just a few references away. To mark these objects, we let GC trace *a small number of references forward* from the red objects, which were retrieved from the PQs. As discussed above, tracing such an object will likely trigger swapping, prepaying the cost for the application to access the object soon later. Note that tracing too many references forward will not be useful, as that may bring in objects not used by the application in the near future. In our implementation, we limit the number of hops to 3, which is often large enough to cover objects in the same logical data structure [132].

After red objects and wavy objects, the remaining live objects to trace are those in Category (3) and the striped objects in Category (1). There are two challenges here. First, there are no easy ways to reach them from the red objects. Second, to reduce memory interference, it is better to trace the striped Category (1) objects before the Category (3) objects, as discussed above.

To tackle these challenges, MemLiner makes every concurrent tracing thread alternate between two modes:

(1) When the PQ is not empty, trace objects in the PQ (*i.e.*, red) and objects a few

references forward (*i.e.*, wavy);

(2) When the PQ is empty, perform normal object-graph traversal that starts from root objects like traditional GC.

Different from a traditional GC, MemLiner modifies the traversal algorithm to consider whether an object o to be traced is likely in local memory (*i.e.*, whether o is a striped Category (1) object or a Category (3) object)—if o is *estimated* to reside in local memory (*i.e.*, a striped Category (1) object), it is traced right away in GC; if not (*i.e.*, Category (3)), MemLiner postpones processing o in its graph traversal until a later time, optimistically hoping that o will be used by the application before it is encountered again in GC. After postponing a number of times (referred to as *MAX_DL* below) , GC processes o even if it is still estimated to be remote, so that the closure computation will not be significantly delayed. MemLiner dynamically adjusts the value of $\boldsymbol{MAX\_DL}$, in response to the size of available heap space. For example, when the available heap size is in the *red zone* (*i.e.*, <15% available space), *MAX_DL* will be set to 0, letting GC quickly finish tracing and collect memory. Details of this adaptive algorithm can be found in this section.

4.3.2.2 Object Location Estimation

Now, the only missing piece of MemLiner's tracing algorithm is a way to *estimate* whether an object is local or not. A naïve solution is to create a system call that allows GC to query the page table. However, this can be prohibitively expensive as it requires a system call per object visited during tracing.

To solve this problem, we conceptually divide the execution into *epochs* and encode the current epoch ID into each *object pointer* whenever an object is accessed. Later on, during concurrent tracing, this epoch ID will allow the GC to estimate how recently an object was accessed and hence how likely it is still in local memory.

Figure 4.5: A 64-bit object pointer in MemLiner.

Epoch. Given our goal of estimating whether an object is in local memory, we define an *epoch* to be an execution period in which the set of pages in local memory that belongs to the JVM process are *relatively stable* (*i.e.*, they do not change much). This set changes as new pages of this JVM process are swapped in and old pages are swapped out. When the change becomes significant (*e.g.*, larger than N% of the total number of JVM pages), a new *epoch* starts. We modify the kernel swap system to keep track of the pages in the cache and determine the start of a new epoch. A global epoch counter is maintained in the JVM and its address is passed into the swap system. This epoch counter starts from zero and is increased by one whenever a new epoch starts.

Timestamp. In the JVM, virtual addresses of objects are represented as *references*, which are essentially pointers with a strong type. In a 64-bit JVM, the format of an object reference is shown in Figure 4.5. Recall that our need is to estimate whether an object is in local memory from a reference/pointer of the object (*e.g.*, recorded in a field of another object) during GC's graph traversal. Our idea here is to modify the pointer format by reserving 4 unused bits as a *timestamp* (*ts* in Figure 4.5) that indicates *the epoch in which the pointer was last dereferenced*—once the epoch ID reaches 15, the next epoch ID goes back to 0. Dereferencing the pointer accesses the target object (*i.e.*, bringing the object to local memory if it is remote). As such, if the timestamp is close to the current epoch, the object is likely in local memory (*i.e.*, Category (1)) and GC should follow the pointer to trace the object; otherwise, the object may not be local (*i.e.*, Category (3)), and GC should postpone tracing it.

Upon the allocation of a new object o, MemLiner sets the timestamp bits in o's pointer to be the current epoch number (with function UpdatePointer in Algorithm 4).

Whenever an object is read/written in an application thread like $b.f = a$ or $a = b.f$

Algorithm 4: Allocation semantics.

Input: Allocation site $o = new\ C$.
Output: Object reference o.

```
1 addr ←ALLOCATE(SIZEOF(C))
2 o ←UPDATEPOINTER(addr, CURRENTEPOCH())
3 return o
```

Algorithm 5: Object read and write semantics in application threads.

Input: Object read/write access $a = b.f$ or $b.f = a$.

```
1 ENQUEUE(PQ, b)
2 b ← UPDATEPOINTER(b, CURRENTEPOCH())
3 if ISREFERENCE(a) then
4     b.f ← a ← UPDATEPOINTER(a, CURRENTEPOCH())
```

(Algorithm 5), MemLiner updates the timestamp ts in the dereferenced pointer b to be the current epoch ID. Furthermore, if a and $b.f$ are also object references, we write an updated pointer of a into $b.f$, indicating that soon the object referenced by $b.f$ will be accessed through a. Again, this instrumentation is implemented through read/write barriers.

Note that we use Algorithm 4 and Algorithm 5 to illustrate the high-level logic. Our implementation actually inserts assembly code for efficiency. Changing object pointers in the JVM would not cause problems for actual memory accesses—although each pointer represents a virtual address, the barriers we use mask pointers so that only the last 42 bits are used to access memory.

4.3.2.3 MemLiner Tracing Algorithm

Algorithm 6 shows GC's tracing logic, which was summarized in §4.3.2.1. The algorithm takes two queue data structures as input: TQ is a standard tracing queue (already used by the JVM) that contains references yet to be explored in object graph traversal; it is initialized with a set of object references in the stack and global variables (*i.e.*, roots). PQ, as discussed earlier, is the producer-consumer queue that contains references of red objects sent to GC by application threads.

Algorithm 6: Main tracing logic in MemLiner's GC.

```
   Input: (1) Producer-consumer queue PQ; (2) tracing queue TQ.
   Output: Fully marked live bitmap for all live objects.
1  Function TRACING(TQ, PQ):
2      while TQ ≠ ∅ do
3          if PQ ≠ ∅ then
4              TRACEREDANDCATEGORY2(TQ, PQ)
5          Tuple ⟨o, dl⟩ ← DEQUEUE(TQ)
6          if DIFF(TS(o), CURRENTEPOCH()) > δ ∧ dl < MAX_DL then
7              ENQUEUE(TQ, ⟨o, dl + 1⟩)
8              Continue
9          if CHECKLIVEBITMAP(o) = 0 then
10             MARKLIVEBITMAP(o)
11             foreach Non-null reference-type field f ∈ o do
12                 Object reference p ← o.f
13                 ENQUEUE(TQ, ⟨p, 0⟩)
```

As discussed in §4.3.2.1, every tracing thread of MemLiner alternates between two modes. In the default mode, tracing loops over the tracing queue TQ, shown in Line 2-13 in Algorithm 6, to perform normal graph traversal. Whenever PQ is not empty (Line 3), the tracing thread interrupts the normal traversal and switches to the other mode to handle the (red) objects in PQ (Line 4); this logic is listed in Algorithm 7 and will be discussed shortly.

In the default mode, each iteration of the tracing loop retrieves a 2-tuple $\langle o, dl \rangle$ from TQ, representing an object reference o and a *delay limit* dl. MemLiner compares TS(o) with the current epoch ID (Line 6). If these two IDs are close to each other (DIFF(TS(o), CURRENTEPOCH()) $\leq \delta$), MemLiner goes ahead to mark this object in the global live bitmap (Line 10) and pushes all the non-null object references stored in this object into the tracing queue TQ (Line 13). Otherwise, MemLiner estimates that the object is not in the cache and hence pushes this tuple back into TQ (Line 7), hoping that the application will use this object and bring it to the cache before the next time it is dequeued in tracing. To avoid pushing back an object too many times, which would delay the completion of closure computation, MemLiner uses a *delay limit* dl, which is initialized to 0. Every time a tuple is pushed back,

Algorithm 7: Tracing logic for red and Category-(2) objects.

```
   Input: (1) Producer-consumer queue PQ; (2) regular tracing queue TQ.
1  Function TraceRedAndCategory2(TQ, PQ):
2      while PQ ≠ ∅ do
3          o ← Dequeue(PQ)
4          Explore(o, TQ, 0)
   Input: (1) Object reference o; (2) tracing queue TQ; (3) current exploration depth depth.
5  Function Explore(o, TQ, depth):
6      MarkLiveBitMap(o)
7      foreach Non-null reference-type field f ∈ o do
8          Object reference p ← o.f
9          if depth < MAX_Depth then
10             Explore(p, TQ, depth + 1)
11         else
12             Enqueue(TQ, ⟨p, 0⟩)
```

its *dl* is incremented (Line 7). Once it becomes *MAX_DL* (*i.e.*, the additional check at Line 6), GC is forced to mark the object. *MAX_DL* is auto-tuned based on the amount of available heap space (discussed shortly).

The other mode of tracing red objects is triggered when PQ is not empty, as illustrated in Algorithm 7. Similar to the default tracing loop, each iteration of the loop (Line 2) in Algorithm 7 retrieves an object reference from PQ, calling a recursive function Explore to not only mark red objects themselves, but also trace a few references forward to mark objects in Category (2), which may be soon used by the application. We use a recursive function here to control the number of references (*i.e.*, data structure depth) to be explored—once *depth* exceeds a constant *MAX_Depth* (Line 9, 3 by default), the function does not further explore the object graph, but instead, pushes these unexplored references into the regular tracing queue TQ (Line 12) so that they can be traced later in a normal graph traversal without priority. This is because, as discussed in §4.3.2.1, following long reference chains can swap in objects that may not be needed by the application in the near future, leading to wasted efforts.

Marking an object live flips its corresponding bit in a global live bitmap (Line 6); as a result, the regular graph traversal (Algorithm 6) would not mark it again if it is encountered there. Once the tracing of the red and Category-(2) objects is done, GC resumes the normal graph traversal in Algorithm 6.

In modern GC with concurrent tracing, each tracing thread works on its own tracing queue TQ. MemLiner modifies each tracing thread to run Algorithm 6 so that the work on TQ is interrupted if there are outstanding red objects in a PQ. Each application thread independently pushes red objects into its thread-local PQ while each tracing thread can consume objects from all PQs. This design makes it possible to enable *work stealing* between threads to balance the number of red and Category-(2) objects processed by these threads. The read/write barrier is already used in existing GC algorithms, such as G1, Shenandoah and ZGC, as well as other far-memory techniques such as AIFM [105]. To further reduce MemLiner's overhead at each read/write barrier, we only need to push the object reference o (64 bit) onto the queue with a very small number of instructions.

Autotuning of *MAX_DL*. How much delay should be introduced to tracing depends on how urgently GC must be completed. As a result, we develop an autotuner that dynamically adjusts the value of *MAX_DL* in response to the available heap size. The rationale is straightforward: if the heap is almost full, there is an urgent need to complete GC and hence we should use a small value for *MAX_DL*; on the contrary, if the heap is mostly available, delaying GC will not have a large impact on memory and hence we use a large value for *MAX_DL* to minimize interference.

MemLiner uses two thresholds for heap availability: 15% and 50%. When the percentage of available memory is lower than 15%, the JVM is in a *red zone*. If the percentage is between 15% and 50%, it is in a *yellow zone*. The JVM is in a *green zone* if the amount of available memory is higher than 50% of the heap size. MemLiner monitors heap usage upon allocations and uses three values for *MAX_DL*: 0, 2, and 4 respectively if the heap falls in the red,

yellow, and green zone. These thresholds were empirically chosen and worked well for all our applications.

4.3.3 Discussion

MemLiner performs adaptation in two dimensions: (1) adapting timestamps based on the swap behavior and (2) adapting *MAX_DL* based on heap availability. The swap behavior correlates with interference and heap availability correlates with GC urgency. We elaborate on how (1) and (2) work in harmony to make MemLiner achieve superior performance.

For (1), MemLiner uses the timestamp mechanism to reduce the interference between GC and application threads. For example, if the cached pages rarely change (*i.e.*, the application has excellent locality or the local memory size is large enough), the interference is minimal and hence it would not create performance issues if MemLiner does not deviate much from an existing GC. Indeed, our algorithm makes the global epoch change slowly and timestamps in most pointers are the same as the current epoch ID. Algorithm 6 would trace most objects in *TQ* without delays. This is a desired property—when resources are *not* constrained, MemLiner would not incur overhead because GC can trace objects and reclaim memory in a timely fashion.

Conversely, if the set of cached pages frequently changes (*i.e.*, the application has poor locality or the cache size is small), the interference is significant and MemLiner should perform differently from an existing GC. Indeed, the global epoch moves at a fast speed. As such, the timestamps in most pointers are different from the current epoch ID. In other words, most objects in the heap are Category-(3) objects that are not in local memory. Consequently, Algorithm 6 would delay the marking of most objects and thus make slow progress. This is also a desired property—tracing should "yield" to the application when local memory resource is tight and application threads are constantly accessing remote memory. In this case, MemLiner imposes a delay to GC, and the delay is bounded by *MAX_DL*.

For (2), we use heap availability to dynamically adjust *MAX_DL*, enabling MemLiner to

"override" the policy made under (1) in urgent situations. For example, if the application is experiencing frequent changes in cached pages (indicating interference) while the heap is almost full, the policy under (1) would delay tracing, which can, in turn, delay the completion of GC and subsequently trigger an undesired full-heap collection. In this case, our adaptation under (2) would determine that the heap is in the red zone and thus change *MAX_DL* to 0—even if tracing is delayed, the delay length is set to 0, effectively allowing GC to move in a normal pace.

4.4 GC-Specific Optimizations

We have implemented MemLiner in both the JVM's default G1 GC [41] and Red Hat's Shenandoah GC [47], which are two representative GCs widely used in cloud settings. G1 is a generational GC that optimizes for throughput with stop-the-world pauses while Shenandoah is a concurrent GC that minimizes the time of each pause by concurrently tracing and compacting objects. Shenandoah optimizes for latency at the cost of reduced throughput. Our goal is to demonstrate that MemLiner can be easily integrated into both GC algorithms, providing performance benefits for different kinds of (*e.g.*, latency-sensitive or batching) workloads.

One challenge in MemLiner is its reliance on read and write barriers, which, if used naïvely, can incur a significant runtime overhead. This section discusses our optimizations to mitigate the overhead. With these optimizations, MemLiner's barrier introduces an average of 2% and 5% overheads, respectively, to Shenandoah and G1, when the application runs entirely with local memory. Such low overheads are due to the following reasons:

First, Shenandoah already utilizes both read and write barriers for concurrent tracing and concurrent evacuation. MemLiner only inserts few instructions into the existing barriers, incurring negligible overheads.

Second, the original G1 only uses the write barrier. Naïvely adding the read barrier into

G1 can cause a much higher overhead. We develop the following three optimizations that successfully filter out a significant fraction of object accesses:

Optimization #1: The enqueue operation of MemLiner's barriers is enabled only when concurrent tracing is in progress. When concurrent tracing is not running, it is unnecessary to add any objects into the PQ.

Optimization #2: G1 is a generational GC that splits the heap into a young and an old generation. Concurrent tracing scans only *old-to-old references* (to compute garbage ratio for each region in the old-gen), meaning that references in the young generation are not traced in concurrent tracing at all. Based on this insight, our read barrier filters out all references in the young generation—there is no need to update their timestamps or add them in PQ because these references are not traced in G1's concurrent tracing anyways.

Optimization #3: Our read barrier does not need to update timestamps for objects whose pointer timestamp is the same as the epoch ID. Essentially, we use a check that first compares the pointer timestamp with the epoch ID and updates the timestamp only if they do not have the same value. The larger the local memory percentage is, the less frequently the epoch changes and hence more objects can benefit from this optimization. This explains why when the percentage of local memory increases, MemLiner's overhead does not increase proportionally (as shown in Figure 4.7).

4.5 Limitations

MemLiner is designed for managed applications running on a managed runtime and thus not applicable to native applications such as those written in C/C++. Furthermore, MemLiner is designed to optimize throughput (by reducing interference and improving prefetching), *not* latency. However, it does not increase the application latency (*i.e.*, making remote access

longer) or the GC pause time. For the Shenandoah GC, its pauses are already very short because operations requiring a pause do not involve many remote accesses and their time is not changed much by MemLiner. For G1, by lining up the tracing and application's memory accesses, MemLiner makes concurrent tracing more efficient, thereby significantly reducing the frequency of triggering full-heap collections. However, it does not reduce the per-collection pause time.

As shown in our evaluation, the more remote memory an application uses, the more effective MemLiner's optimization. However, when a large percentage of the working set fits into local memory, MemLiner's effectiveness reduces. In fact, if this percentage exceeds 50%, MemLiner's performance is on par with that of the original JVM.

The other limitation is that MemLiner focuses on reducing interference between the application and concurrent tracing threads. Application threads may also interfere with memory reclamation threads if the GC performs concurrent reclamation (such as Shenandoah and ZGC). MemLiner cannot reduce this type of interference.

4.6 Evaluation

4.6.1 Experiment Setup

We implemented MemLiner on top of OpenJDK 12 (v 12.0.2) and Linux (v 5.4.0). Our swap system is based upon our re-implementation of FastSwap [14][2], which provides good swap performance. We implemented it on top of G1 and Shenandoah. Implementing MemLiner in other GCs would be straightforward in the future.

Environment. We ran our experiments with two machines, each with two Xeon(R) CPU E5-2640 v3 processors, 128GB memory, one 1TB SSD, and one 40 Gbps Mellanox ConnectX-3

[2]Its original implementation was incompatible with OpenJDK12.

Spark [135]	**Dataset**	**Size**
MLlib KMeans (**SKM**)	Wikipedia France [7]	1.1GB
Spark Linear Regression (**SLR**)	Wikipedia English [7]	3GB
Spark Transitive Closure (**STC**)	Synthetic graph	1.5M edges 384K vertices
Cassandra [8]	**Workload**	**Operation**
Update Intensive (**CUI**)	Update 50% Insert 50%	10M ops
Read Intensive (**CRI**)	Read 50% Insert 50%	10M ops
Insert Intensive (**CII**)	Insert 50% Update 25% Read 25%	10M ops
Neo4j [9]	**Dataset**	**Size**
PageRank (**NPR**)	Wikipedia Turkish [7]	14M edges 544K vertices
Triangle Counting (**NTR**)	Wikipedia Turkish [7]	14M edges 544K vertices
Degree Centrality (**NDC**)	Dogster Friends [7]	8.5M edges 451K vertices
QuickCached [5]	**Workload**	**Operation**
Write Dominant (**QWD**)	Insert 60% Read 40%	9M ops
Read Dominant (**QRD**)	Insert 20% Read 80%	9M ops
DayTrader [61]	**Workload**	**Size**
Tradesoap (**DTS**)	Synthetic set of stocks	12288 users 8192 sessions

Table 4.1: Applications and datesets used for G1.

InfiniBand network adapter. They are connected by one Mellanox 100 Gbps InfiniBand switch. One machine runs the JVM process while the other provides remote memory via RDMA. All our experiments used a 32GB heap and 4K pages.

Although our application heap size is relatively small (compared to the size of main memory on our machines), the performance of a remote-memory application depends on how much of its working set can fit into local memory and how many (application and GC) threads are used, *not* on how large local memory is. In particular, MemLiner's key data structure is a per-thread PQ (*i.e.*, TQ is not key to MemLiner as it is GC's original data structure). PQ's size depends on the ratio between the number of applications and the number of tracing threads. For instance, for G1, we follow Oracle's recommendation [95] by setting the number of parallel GC threads to be $5 \times (\text{core number})/8$, and the number of concurrent tracing threads to be 1/4 of the parallel GC threads. With this ratio and a per-thread PQ of 1024 entries, we rarely saw overwrites in our experiments (with our filtering optimizations stated above). However large the heap is, as long as this ratio remains the same, the size of PQ does not need to change; so does the work done by MemLiner.

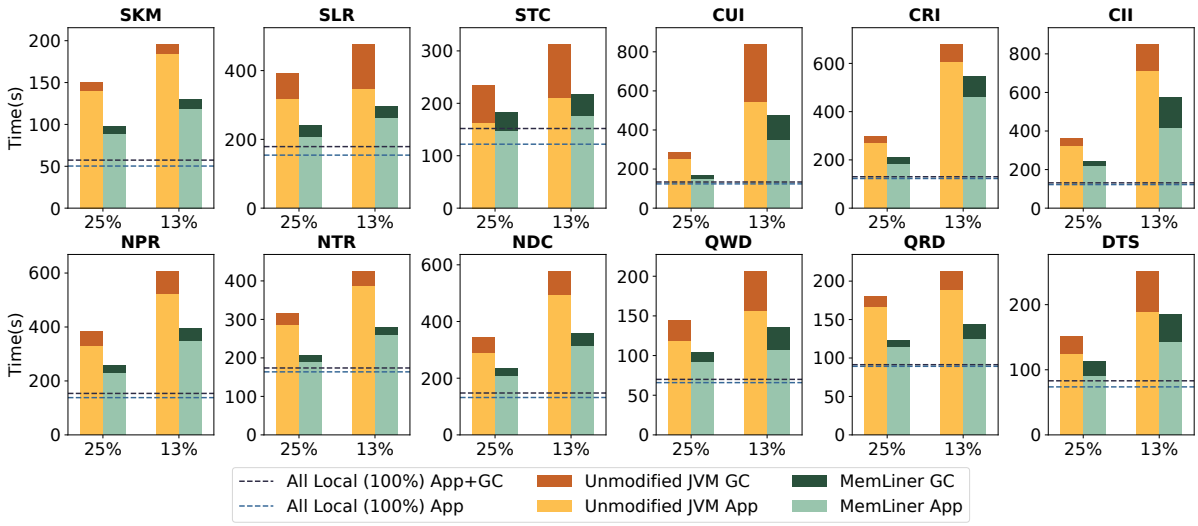


Figure 4.6: Performance comparisons between G1 GC (yellow bars) and MemLiner (green bars) under two local memory ratios: 25% and 13%; each bar is split into application (bottom with light colors) and GC (top with dark colors) time in seconds. The two dashed lines show application time and total time with unmodified JVM and 100% local memory (no swaps).

Applications. To evaluate MemLiner, we used a range of cloud applications including Apache Spark [135] (3.0.0), the de-facto data analytics system, Apache Cassandra [8] (3.11), a widely used distributed database, Neo4j [9] (4.3.2), a graph database, QuickCached [5], a Java implementation of Memcached, as well as DayTrader [61], IBM's open-source application emulating an online stock trading system. These applications cover a wide spectrum of text and graph analytics, web services, machine learning tasks, and database query tasks. For each application, their workloads and datasets are reported in Table 4.1.

The memory access patterns of our applications can be categorized into three types:

- *Mostly sequential access patterns*: Spark applications operate over RDDs. An RDD is an object array or serialized primitive array. Each application thread exhibits clear memory access patterns, *e.g.*, streaming or stride.
- *Random access patterns*: QuickCached (a key-value store) and DayTrader (stock trading simulation) exhibit quite random memory access patterns.
- *Mixed access patterns*: Take Cassandra as an example. Each read/update operation goes

Local Memory	G1 GC			Shenandoah GC		
Configuration	**App**	**GC**	**All**	**App**	**GC**	**All**
25% Local	**1.45×**	**1.65×**	**1.48×**	**1.88×**	**15.33×**	**2.16×**
13% Local	**1.46×**	**1.79×**	**1.51×**	**1.60×**	**6.20×**	**1.80×**

Table 4.2: Speedups provided by MemLiner for G1 and Shenandoah. (speedup: the average time under each configuration using the unmodified JVM divided by that using MemLiner)

through several micro-operations. Different micro-operations have different memory access patterns, *i.e.*, the MemTable loading exhibits a good streaming memory access pattern and some other calculations access memory randomly. Both Cassandra and Neo4j belong to this category.

Our experiments considered two local memory ratios: 25%, and 13% of the total Java heap size (32GB), which are consistent with local memory ratios used in prior work [105, 125]. We enforced these ratios with `cgroup`.

4.6.2 Performance with G1 GC

Overall. Figure 4.6 compares the performance of the baseline (the default G1 GC) and MemLiner under two different local memory ratios: 25%, and 13%. As shown, MemLiner offers better performance over the baseline JVM for all workloads, **1.48×** speedup on average under 25% local memory and **1.51×** speedup on average under 13% local memory. A summary of these performance improvements (for the application, GC, and end-to-end performance) is reported in Table 4.2.

We also compared the number of swap-in pages between MemLiner and the unmodified JVM: MemLiner reduces an average of **81%** of on-demand swap-ins and **56%** of total swap-ins (including both on-demand and prefetching swaps).

Compared with running the whole application in local memory with no swapping (illustrated by dashed lines in Figure 4.6), the unmodified JVM incurs 2.17× and 3.73× slowdowns under the 25% and 13% local memory configurations, respectively. MemLiner brings them down to 1.47× and 2.48×.

Details. For several workloads (e.g., **SLR**, **STC**, **CUI**, **NDC**, **QWD** and **DTS**), the default JVM's GC time increases dramatically when the local memory ratio drops from 25% to 13%. This is because when memory resources are tight, concurrent tracing becomes slow with many local-memory cache misses. It sometimes cannot finish a complete closure before the heap is full, causing the JVM to pause all application threads and run a time-consuming full-heap GC. Fortunately, MemLiner brings down that GC cost, enabling concurrent tracing to quickly compute the closure by following the applications' accesses and reducing full-heap GCs.

Cassandra's performance degrades drastically under 13% local memory. In addition to more frequent full-heap GCs, this also stems from data spilling. When the memory usage exceeds a certain ratio (*e.g.*, 2/3) of the heap size, Cassandra automatically spills data from memory to disk. Since concurrent tracing under a tighter local-memory budget becomes much slower, the memory consumption frequently exceeds that ratio, triggering spilling and slowing down the application. In these large-scale systems, GC can actually impact the performance of applications in many unexpected ways.

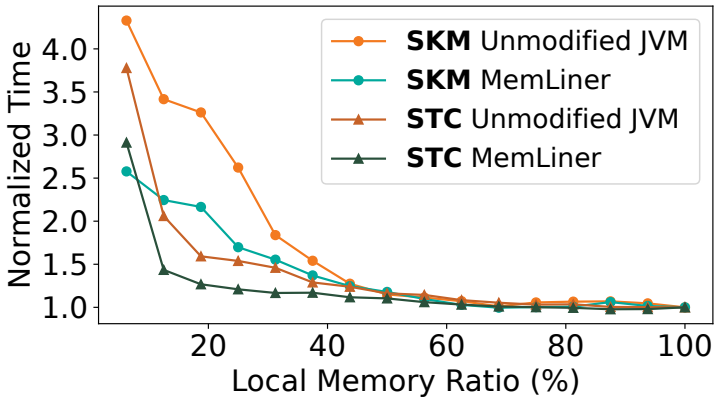


Figure 4.7: Performance comparisons for **SKM** and **STC** between the unmodified JVM and MemLiner under different local memory configurations.

Different Local Memory Configurations. We ran **SKM** and **STC** with various local-memory ratio configurations and report the performance in Figure 4.7. As shown, the lower

the ratio, the higher the benefit MemLiner provides. For both applications, the turning point is around 50%—MemLiner and the baseline have about the same performance when the local memory ratio reaches 50% or above.

4.6.3 Performance with Shenandoah GC

To demonstrate the generality of MemLiner, we implemented MemLiner in a second garbage collector: Shenandoah[47], a widely-used highly-concurrent low-pause GC developed by Red Hat. It performs not only concurrent tracing but also concurrent object evaluation to minimize pauses.

Shenandoah provides great latency benefits under sufficient local memory. However, it has extremely poor performance with remote memory involved. For example, the slowdowns under 25% memory for our Spark and Neo4j applications are constantly above 10× and 4×, respectively. Compared to Neo4j, Spark applications usually have much larger working sets, leading to more remote accesses. Such a large overhead highlights the problem of running many concurrent GC threads that do not align with the application's memory access. In particular, Shenandoah is *not* a generational GC (while G1 is). In G1, when the young generation, which contains short-lived objects, is full, the JVM suspends application threads and evacuates objects in the young generation. This leads to excellent data locality after evacuation. However, under Shenandoah GC, the JVM runs concurrent tracing much more frequently to scan the full heap to identify and collect garbage. Those tracing threads exhibit particularly poor locality. To evaluate Shenandoah, we had to use smaller datasets (Table 4.3) for a tolerable running time.

As illustrated in Figure 4.8 and summarized in Table 4.2, MemLiner achieves an overall **2.16**× and **1.80**× speedup compared to the unmodified JVM under 25% and 13% local memory, respectively. MemLiner reduces an average of 82% on-demand swap-ins and 56% of total swap-ins under 25% local memory, while it reduces 79% of on-demand swap-ins and 22% of total swap-ins under 13% local memory. As shown in Table 4.2, MemLiner provides

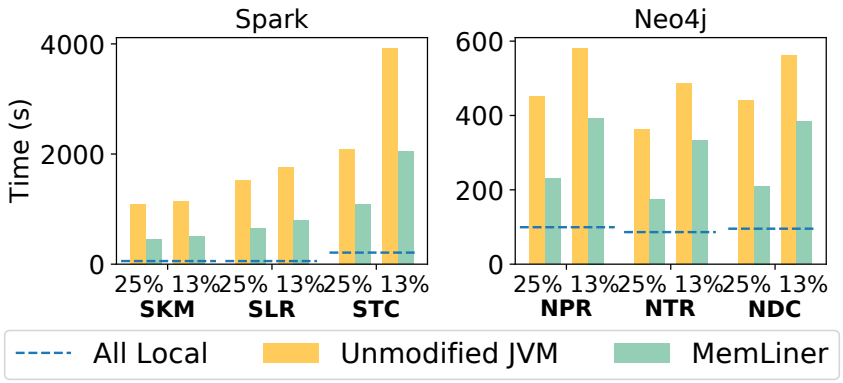


Figure 4.8: Performance comparison with Shenandoah GC [47].

Spark Programs	Dataset	Size
MLlib KMeans (**SKM**)	Wikipedia Polish [7]	1GB
Spark Linear Regression (**SLR**)	Wikipedia Polish [7]	1GB
Spark Transitive Closure (**STC**)	Synthetic Graph	1.5M edges 384K vertices
Neo4J Programs	**Dataset**	**Size**
PageRank (**NPR**)	Wikipedia Slovak [7]	7.6M edges 291K vertices
Triangle Counting (**NTR**)	Wikipedia Slovak [7]	7.6M edges 291K vertices
Degree Centrality (**NDC**)	Wikipedia min-nan [7]	4.4M edges 429K vertices

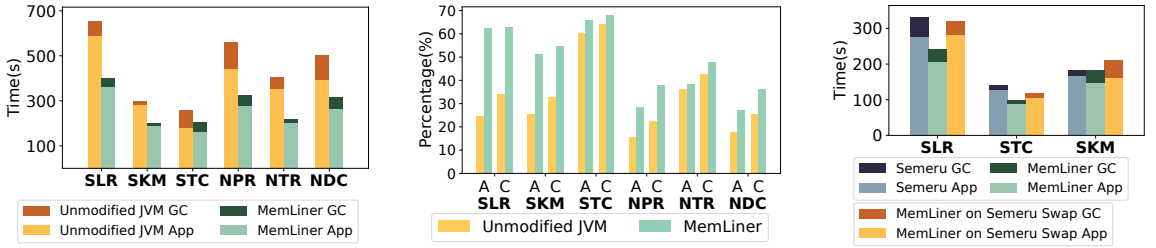
Table 4.3: Benchmarks and datasets for Shenandoah.

tremendous improvements for Shenandoah's GC performance, because the unmodified JVM frequently triggers *full-heap stop-the-world GC*.

4.6.4 Comparisons with Other Systems

Leap [81] is an advanced OS-level prefetcher. It uses a major-vote algorithm to determine how to do prefetches. In cases where no clear access patterns are seen, Leap aggressively prefetches consecutive pages. Although this strategy may improve performance for native applications whose memory accesses often fall into large arrays, it often hurts managed applications such as Spark, as GC's pointer-chasing behavior often makes prefetched consecutive pages useless.

Our hypothesis is that even aggressive prefetchers like Leap cannot handle the interference of GC, and that by aligning the memory accesses of GC with application threads, MemLiner



(a) Leap: Running time (b) Leap: Accuracy(A) and Coverage(C) (c) Semeru: Running time

Figure 4.9: Performance comparisons with Leap and Semeru; Semeru crashed on **NPR**, **NTR**, and **NDC** (*i.e.*, Neo4j applications).

can improve application performance under Leap just like under less aggressive prefetchers. To test our hypothesis, we compared MemLiner with the unmodified JVM (default G1 GC) both using Leap as the prefetcher. This experiment was conducted on three Spark applications: **SLR**, **SKM**, **STC**, and three Neo4j applications: **NPR**, **NTR**, **NDC**, under 25% local memory.

As shown in Figure 4.9(a), compared with the unmodified JVM on Leap, MemLiner improves the overall performance by an average of 1.6× and reduces 58% of on-demand swap-ins, as well as 53% of total swap-ins on average. To understand whether MemLiner improves Leap's prefetching effectiveness, we additionally measured Leap's prefetching *accuracy* (i.e., the percentage of page faults hitting on the swap cache among prefetched pages) and *coverage* (i.e., the percentage of swap cache hits among all page faults) with and without MemLiner. As shown in Figure 4.9(b), MemLiner helps Leap deliver higher accuracy and coverage. We still observed that MemLiner is not as useful for **STC** and **NTR** as it is for the two applications. This is because the number of live objects in **STC** during concurrent tracing is relatively small, leading to shorter tracing time and better access patterns. For **NTR**, its application threads exhibit random memory accesses themselves. Hence, Leap cannot detect clear patterns even if MemLiner has already eliminated much of the interference.

Semeru [125] is a memory-disaggregated runtime, where the entire Java heap is backed by physical memory on memory servers and the CPU server's local memory is used as an

inclusive cache. Semeru completely redesigned the JVM so that all the garbage collection is offloaded from the CPU server to the memory servers, through special lightweight JVMs running there. Applications execute on the CPU server with absolutely no GC interference, at the cost of extra computation on memory servers (i.e., two extra cores for each memory server to run the offloaded lightweight JVM).

Here, to evaluate whether MemLiner can achieve similar performance as Semeru, without Semeru's intrusive changes to JVM and Semeru's extra computation load on memory servers, we ran the same three Spark applications under 25% local memory on top of (1) Semeru, (2) MemLiner on Semeru's swap system (*i.e.*, a modified version of NVMe-over-fabrics [1]), and (3) MemLiner on FastSwap [14], which is the default swap system MemLiner builds on. We ran Semeru with one CPU server and two memory servers—the Java heap is partitioned between the memory servers.

As shown in Figure 4.9(c) , MemLiner's performance is comparable with Semeru when using Semeru's swap system, and is much better than Semeru when using MemLiner's default swap system. The reason is that, even though Semeru completely eliminates GC tracing threads from the local machine, it has to perform a great deal of coordination between servers to handle cross-server references, incurring communication overheads. We would have also liked to run Semeru directly over FastSwap, but this was not feasible due to Semeru's runtime-kernel co-design that prevents Semeru from easily adapting to different swap systems.

We could not directly compare Memliner with AIFM [105] as AIFM targets native languages (C/C++) applications and requires rewriting programs. However, the major idea behind AIFM—swapping at the object granularity—is orthogonal to MemLiner. MemLiner can also benefit from a redesigned swap system that performs object-level swapping.

4.6.5 More Detailed Results

Memory Reclamation Impact. Since MemLiner postpones tracing objects estimated to be remote, it may delay memory reclamation. To understand the impact of such a delay,

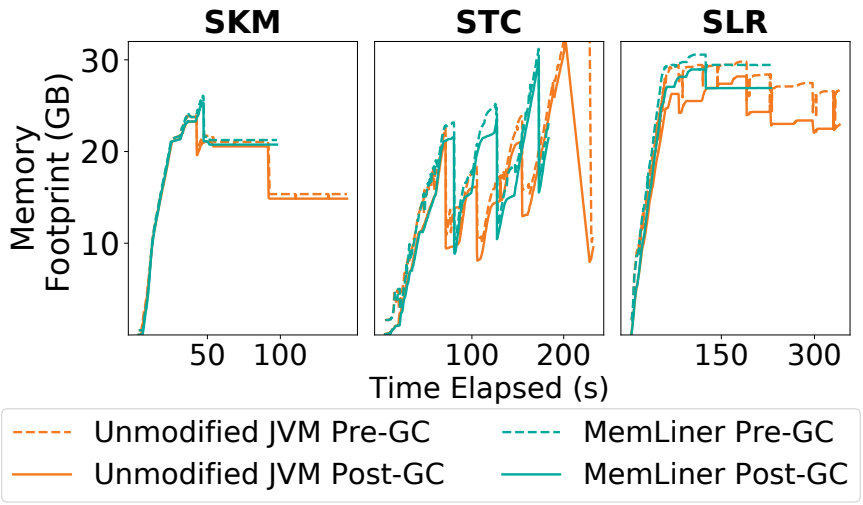


Figure 4.10: Memory footprints for **SKM**, **STC**, and **SLR**, between unmodified JVM and MemLiner under 25% rate.

we collected post-GC memory footprints for **STC**, **SKM**, and **SLR** executed atop the unmodified JVM and MemLiner under 25% local memory configuration. Figure 4.10 reports, for each program, both its pre-GC and post-GC memory footprints. As shown, for all three workloads, MemLiner incurs insignificant delays in memory reclamation and only a slight increase in the peak memory consumption. This is because tracing of each remote object can only be postponed a few times (*i.e.*, *MAX_DL*); when the available heap runs low, *MAX_DL* becomes 0 and we do not postpone GC at all.

Epoch Estimation Effectiveness. We collected the number of objects that are scanned from PQ and TQ for three Spark applications under 25% local memory. The ratio of objects scanned from PQ over total objects scanned during the concurrent tracing phase is 45%, 42%, and 11% respectively for **SLR**, **SKM** and **STC**. We also evaluated MemLiner after disabling epoch estimation: we saw an overall performance degradation of 8.6%, 8.8% and 11.3% respectively, for **SLR**, **SKM** and **STC** under 25% local memory.

4.7 Summary

This chapter presents MemLiner, a runtime technique that reduces the GC-application interference by aligning the memory accesses of application and tracing threads. We classify reachable objects into three categories and treat objects in each category in a different way to achieve the two seemingly conflicting goals. Our promising results with two production GCs demonstrate that MemLiner can be readily used in today's datacenters.

CHAPTER 5

Language-Guided Distributed Shared Memory with Ultra Efficiency

In a fully disaggregated datacenter, applications can leverage multiple compute servers, thereby harnessing greater computational power. This distribution of application threads across different servers necessitates a shared memory abstraction known as Distributed Shared Memory (DSM). The concept of distributed shared memory (DSM) received significant attention during the early years of distributed computing systems. This era witnessed a plethora of pioneering efforts, as exemplified by seminal works such as [21, 30–32, 46, 54, 74, 75, 83, 91–93, 118].

DSM offers the power of parallel computing using multiple processors and machines and, more crucially, streamlines the development of distributed applications with a unified, contiguous memory view. It was initially greeted with enthusiasm, but significant performance bottlenecks emerged, mainly due to the limited network speeds at the time. Recent advances in hardware and networking technologies, such as [13, 17, 24, 34, 39, 44, 49, 56, 58, 63, 69, 76, 82, 94, 97, 108, 115], have renewed interest in DSM. Several new DSM systems have been proposed to leverage these advanced networks, including [27, 68, 88, 112, 119, 128]. Despite these developments, these systems still struggle to achieve satisfactory performance, with poor scalability and significant slowdowns compared to single-machine setups. This is largely due to the high synchronization overhead required to maintain memory consistency across multiple servers.

This chapter introduces an efficient DSM implementation based on the insight that the

ownership model embedded in programming languages such as Rust automatically constrains the order of read and write, providing opportunities for significantly simplifying the coherence implementation if the ownership semantics can be exposed to and leveraged by the runtime. This chapter discusses the design and implementation of DRust, a Rust-based DSM system that outperforms the two state-of-the-art DSM systems GAM [27] and Grappa [89] by up to 2.64× and 29.16× in throughput.

5.1 Introduction

The concept of distributed shared memory (DSM) received significant attention during the early years of distributed computing systems. This era witnessed a plethora of pioneering efforts, as exemplified by seminal works such as [21, 30–32, 46, 54, 74, 75, 83, 91–93, 118]. DSM offers the power of parallel computing using multiple processors and machines and, more crucially, streamlines the development of distributed applications with a unified, contiguous memory view.

The initial enthusiasm for DSM was tempered by significant performance bottlenecks, primarily due to the low network speeds prevalent during its nascent stages. Recent advances in hardware and networking technologies [13, 17, 24, 34, 39, 44, 49, 56, 58, 63, 69, 76, 82, 94, 97, 108, 115] have revitalized the DSM explorations. Several new DSM systems [27, 68, 88, 112, 119, 128] were proposed in recent years to take advantage of these enhanced networks. However, these systems are still far from achieving satisfactory performance, exhibiting poor scalability and substantial slowdown compared to their single-machine counterparts. This is mainly due to the intensive synchronization operations needed to ensure memory coherence across servers.

State of the art. The majority of existing DSM systems [16, 27, 68, 128] adopt an approach to achieve data consistency by adhering to the following invariant: for each data

block to be accessed, the block is either located on a single node with potential read and write access, or it is replicated across multiple nodes with each having read access only. Prior to a server attempting to access a block, a DSM system checks the state of the block, invalidates copies of that block on all other servers, and then transmits the block to the requesting server. This synchronization process necessitates multiple network round trips. Even with RDMA, the incurred latency is still orders of magnitude higher compared to a single local access, significantly degrading overall performance. Effectively reducing the number of synchronizations is, therefore, crucial for minimizing DSM overhead and rendering it feasible for real-world deployment.

A practical strategy to minimize synchronization overhead involves implementing high-level protocols to guarantee exclusive access for each server. For instance, Apache Spark [135] utilizes an immutable data structure known as a resilient distributed dataset (RDD) for distributed access. However, RDD only facilitates coarse-grained distributed access, limiting each server to accessing a distinct partition of an RDD. While increasing access granularity enhances performance, it comes at the expense of reduced generality—Spark is tailored for bulk processing of batch data and is incapable of supporting distributed applications requiring object-level accesses, such as social networks where objects of various types and sizes (*e.g.*, images, connections, *etc.*) are created and manipulated upon each user request.

Insights. Our main observation is that synchronization overheads in existing DSM systems are introduced primarily due to the use of a generic approach that overlooks semantic information from programs. For example, many real-world concurrent programs are engineered with a single-writer-multiple-reader (SWMR) discipline to ensure correctness during concurrent operations. Leveraging such information can potentially eliminate the need to check the state of remote data blocks before accessing them, leading to dramatically improved performance. A major challenge is, however, how to expose such semantics in a sensible way so that the DSM system can see and act upon it.

One approach to convey such semantics, as demonstrated by AIFM [106] and Midas [102], involves exposing APIs that developers can invoke to specify program regions accessible only by a single writer. However, this process is cumbersome and error-prone, demanding a profound understanding of potential executions and involving substantial program writing. Our key insight in this endeavor is that the SWMR programming paradigm aligns seamlessly with *ownership types*, which have already been integrated into programming languages like Rust [109]. Rust is widely employed in the system community for dependable and secure implementation of low-level systems code.

Rust's ownership type inherently upholds SWMR properties in any compiled Rust program. The fundamental concept behind the ownership type is that each value is ensured to have a single unique variable as its owner throughout the execution. While multiple references to a value are allowed, only the owner and mutable references can modify the value. Moreover, only one of these references is permitted to be used for modifying the value at any given point.

When developing a DSM system on top of an ownership-based language like Rust, SWMR semantics are inherently embedded in any Rust program *by design*. Effortlessly extracting such information becomes possible with basic compiler support, sparing developers from the need for code rewriting. Utilizing the SWMR semantics from the program leads to a considerably simplified process for accessing data in DSM. In the case of a write access, the ownership type ensures exclusive access to the data. Consequently, DRust can move the data to the requesting machine, performing the write there without explicitly invalidating its copies on other machines. In the case of a read access, data can be efficiently replicated to (and cached in) each requesting machine, benefiting from the compiler-provided assurance of freedom from concurrent writes.

This paper presents DRust, an efficient Rust-based DSM implementation that enables object-level concurrent accesses by leveraging the SWMR semantics made explicit by Rust's ownership type. DRust automatically turns a single-machine Rust program into a DSM-based

distributed version *without requiring code rewriting*. While extracting the ownership semantics appears straightforward, leveraging it to implement a distributed coherence protocol correctly and efficiently presents two main challenges.

The first challenge is *how to manage memory correctly and efficiently*. Rust's ownership type system is inherently designed for a single-machine environment, where the memory address of an object remains constant post-creation. This assumption is disrupted in a distributed environment, where objects may be migrated or duplicated on different machines. Such actions can lead to the risk of dangling pointers, potentially breaking memory coherence.

To tackle these issues, DRust builds a global heap spanning multiple servers based on the idea of partitioned global address space [37]. Each object in the heap has a unique global address in the address space, which can be used for accessing the object from any server. DRust re-implements Rust's memory management constructs to allocate objects in the global heap. Given that a server can have cached objects (to accelerate reads), DRust carefully crafts an ownership-based cache coherence protocol upon the global heap abstraction to achieve both memory coherence and efficiency (§5.3.1.1).

In a nutshell, our coherence protocol leverages the ownership semantics to eliminate the need for explicit cache invalidation. It allows multiple readers to fetch a copy of the object from its host server and cache it, but disallows any change to the global address and the value of the object. When a write access occurs, it must first borrow the ownership, at which point DRust moves the object in the global heap to a new address on the server issuing the write. The address change of the object automatically invalidates cache copies that use the stale address and triggers the subsequent readers to update the cache by fetching the object from its latest address.

The second challenge is *how to support transparency in programming*. Rust's standard libraries and programs were originally built for running on a single machine, and they cannot deal with distributed resources in a cluster. For example, a Rust program running on server A cannot spawn a thread on another server B, let alone synchronize threads between

A and B. To enable a Rust program to run *as is* under DRust, we provide distributed threading utilities by restructuring critical elements of the Rust standard library, including threading, communication channels, and shared-state locks (§5.3.1.2). Our adapted libraries offer the same interfaces, making them compatible with single-machine Rust programs, but internally invoke our distributed scheduler, which determines where to run the thread and facilitates cross-server synchronization. We built them atop the ownership-based memory model, enabling the DRust runtime to safely pass references of objects between threads and automatically fetch the value from the global heap upon dereferencing.

With our programming abstractions, a Rust application can start on a single server and gradually spawn its threads to other servers. Under the hood, DRust employs a runtime to manage distributed physical compute and memory resources for the application. The runtime runs as a process on each node in the cluster, and they work cooperatively for cross-server memory allocation and thread scheduling. The runtime prioritizes the current server for object allocation and thread creation, but it will schedule the resource allocation request to another server under memory pressure (§5.3.2.1). To make cluster-wise decisions such as deciding the target server for global memory allocation and thread creation, DRust has a global controller that is launched together with the application. The global controller communicates with DRust runtime on each node to collect resource usage information and applies adaptive policies to achieve load balance (§5.3.2.2).

Results. We evaluated our system on four real-world applications in an eight-node cluster. Our evaluation demonstrated an average of 2.02× and 9.48× (up to 2.64× and 29.16×) speedup compared with two state-of-the-art DSM systems GAM and Grappa, respectively. Furthermore, DRust incurred a mere 2.42% slowdown compared to the original Rust program on a single machine with sufficient resources.

5.2 Motivation

DSM was proposed to eliminate the barrier of distributed programming by offering the same memory consistency model as single-machine shared memory. The core of its design is a software-based cache coherence protocol, which mimics a hardware-based approach on multi-core CPUs and synchronizes memory states on different servers by sending control messages between them. However, it is notoriously hard to implement cache coherence efficiently at the software level due to the high communication latency between physically disjointed servers.

High Synchronization Overheads for Coherence. To gain a high-level understanding of how much improvement can be achieved by improving the cache coherence protocol, we performed an analysis by running a real-world application DataFrame [99] with a state-of-the-art DSM system GAM [27] with a fast network. We first ran Dataframe on a single server with 16 CPU cores and 64GB memory. We then ran it with GAM on eight servers connected by a 40Gbps Infiniband network by evenly distributing the same amount of resources to eight servers (*i.e.*, each server uses 2 CPU cores and 8GB memory). Our experiments show a **2.4×** slowdown when Dataframe runs on eight servers.

A detailed examination reveals that such a slowdown stems primarily from its complicated coherence protocol. GAM runs a directory-based protocol, which assigns each DSM cache block a home node. Upon each object read/write, the home node tracks the state of its cache block and updates all cache copies for the state change, incurring extensive computation and network overhead. We broke down the average time spent on each component when accessing one object in the DSM. Reading a 512-byte (*i.e.*, GAM's default cache block size) uncached object in GAM takes 16µs, while the actual time to read the object over the network is only 3.6µs. In other words, maintaining cache coherence takes **77%** of the total time. This large memory access overhead significantly increases operation latency, hindering the practical deployment of distributed shared memory. With the single writer invariant inherent

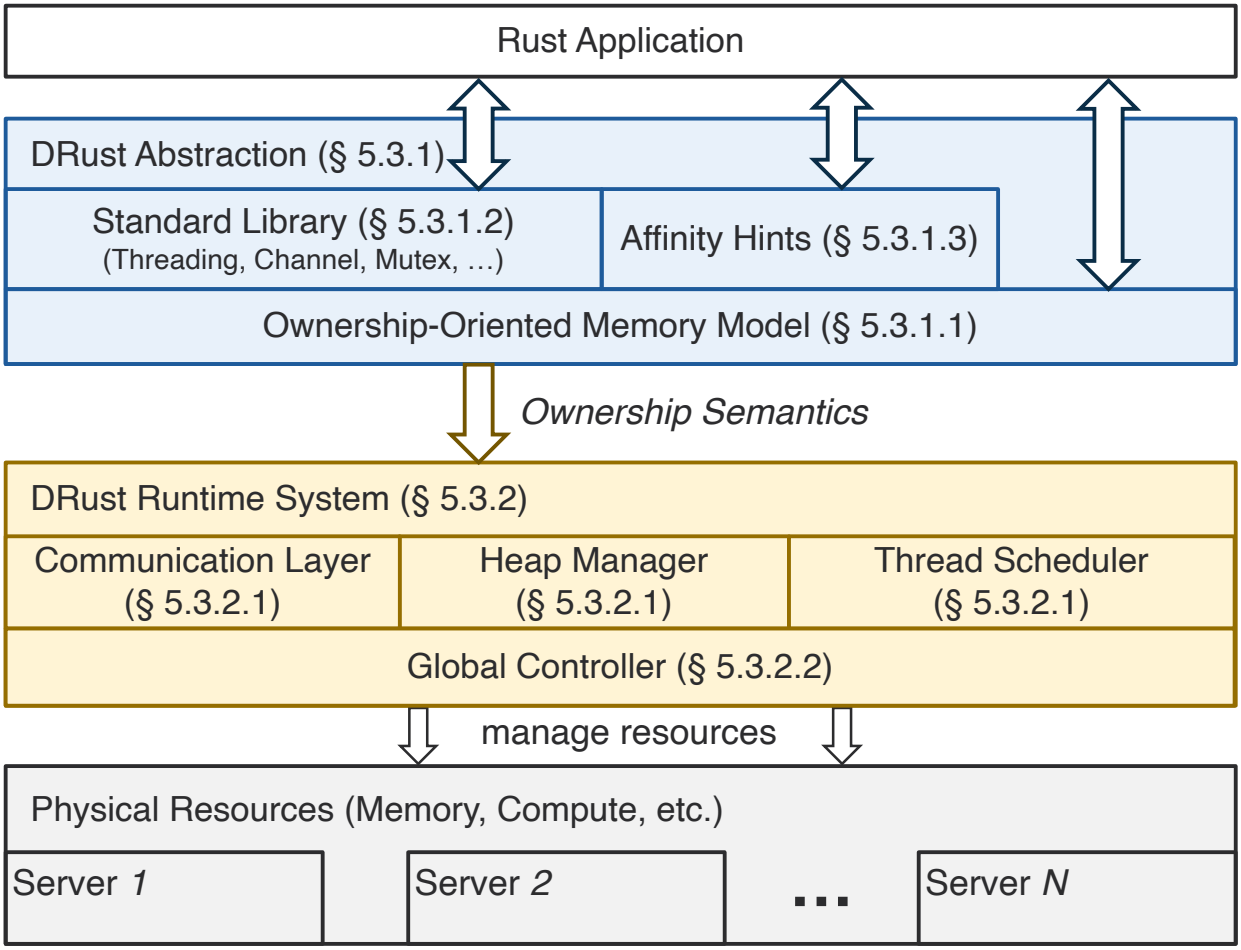


Figure 5.1: Design overview of DRust.

in the ownership model, we expect that most of this overhead can be eliminated, leading to significant (> 2×) speedups for each access.

5.3 Design

DRust is an efficient DSM framework atop the Rust programming language. As shown in Figure 5.1, it consists of Rust-based programming abstractions for DSM (§5.3.1) and a runtime (§5.3.2) that manages distributed physical resources.

DRust is compatible with standard Rust. Listing 5.1 illustrates how the accumulator (shown in Listing 2.1) runs on DRust distributively without requiring code rewriting. The

```
1 // Unmodified Rust code.
2 pub struct Accumulator { pub val: Box<i32>, }
3 impl Accumulator {
4   pub fn add(&mut self, delta: &i32)->i32 {
5     *self.val += *delta;
6     *self.val
7   }
8 }
9 fn main() {
10   // Allocates two integers in the distributed heap.
11   let val: Box<i32> = Box::new(5);
12   let b: Box<i32> = Box::new(10);
13   let mut a = Accumulator{val};
14   // a.val and b will be fetched to local.
15   let local_add = a.add(&*b); // a.val == 15
16   // Only refs to a and b are shipped to remote.
17   let remote_add = thread::spawn(move ||
18     a.add(&*b)).join(); // a.val == 25
19 }
```

Listing 5.1: DRust seamlessly transforms an unmodified accumulator implemented in Rust into a distributed version.

program starts running on a single machine A and the DRust runtime gradually allocates its memory and spawns new threads on different machines. Specifically, Lines 10–13 create `Accumulator a` and `b` where `a.val` and `b` are in the global heap. We use a global allocator to allocate objects in the global address space and hence these objects may be allocated on a different server. Line 15 synchronously adds `b` to `a` by fetching both values `a.val` and `b` to A's local memory (if they are allocated somewhere else). Line 17 spawns a new thread and ships the function closure to perform `add` asynchronously. This thread will be scheduled on a different server B if A's compute power has been saturated. In this case, DRust performs shallow copying and only ships the pointers stored in `a` and `b` to B without actually moving objects in the global heap. The newly-created thread relies on the DRust runtime to detect data locations and fetch objects upon dereferencing.

5.3.1 DRust Programming Abstraction

DRust provides each thread with a local stack and abstracts distributed memory as a shared global heap. Each server allocates thread stacks and backs one partition of the global heap with its physical memory. DRust re-implemented core memory management constructs including

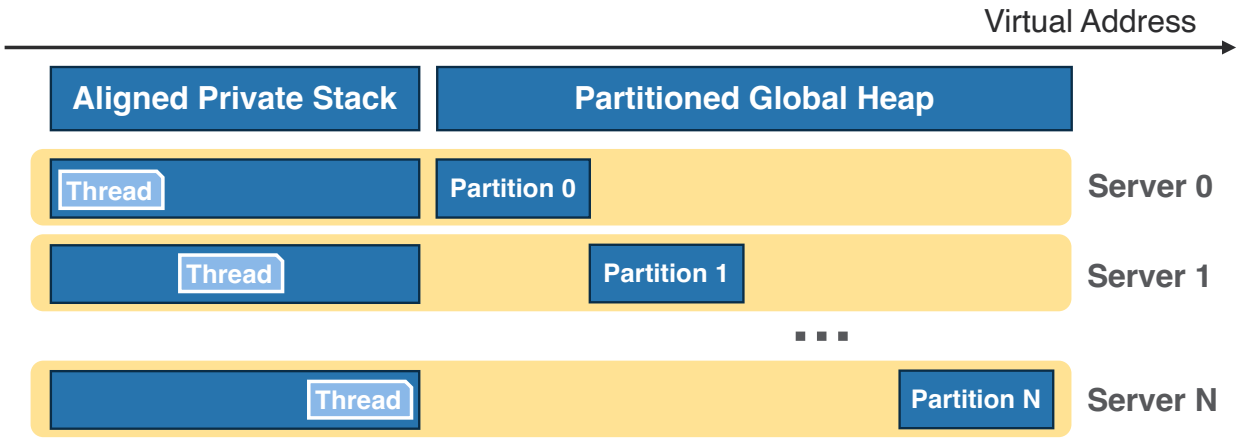


Figure 5.2: The address space layout of DRust. The stack is private to each thread but they share an aligned address space to ease migration, while the heap is globally shared and partitioned across servers.

`Box`, `&`, and `&mut` for transparent heap access. This approach hides the complex details of memory allocation/deallocation, moving objects, and coherence maintenance (§5.3.1.1). DRust supports distributed threading and synchronization by adapting Rust's standard libraries atop the core language constructs (§5.3.1.2). Furthermore, DRust offers affinity annotations that allow developers to build more efficient applications by expressing data affinity semantics (§5.3.1.3).

5.3.1.1 Memory Management

Next, we discuss how DRust (re)implements the memory-related language constructs in Rust to achieve memory safety and memory coherence.

Address Space. As shown in Figure 5.2, DRust maintains an identical address space layout on all servers. It exposes distributed memory as a coherent shared heap to applications. Embracing the idea of partitioned global address space (PGAS) [37], it partitions the heap space and assigns each server a unique address range. The stack, in contrast, is private to each thread. However, DRust aligns the stack space on each server and pads stacks to avoid overlapping. This streamlines thread migration between servers as it allows a thread to keep

its private stack address unchanged when being moved.

Coherence Protocol in a Nutshell. For efficiency, DRust employs a *call-by-reference* model for newly created threads. Upon creation of a thread, the DRust runtime only passes references or `Box` pointers to objects to the newly created thread. Upon dereferencing, objects are fetched to the server where the thread is executed.

When a *read access* of an object is issued on a server, our runtime simply fetches a copy of the object from its hosting server and places it in its *local cache*. As a result, multiple copies of the same object may exist on different servers. This allows multiple servers to read the object at the same time from their respective cached copies. Fetching a copy of the object for read does not change the object's address in the global space. When a *write access* occurs on an object, the server issuing the write must first obtain the object's write access permission through a *mutable borrow*. Our reimplementation of mutable borrow (discussed shortly) *moves*[1] *the object in the global heap to a new address* that belongs to that server. In doing so, the object's cached copies on other servers are automatically invalidated—subsequent reads on these servers must obtain an immutable reference to the object through an immutable borrow from its owner pointer, which has been updated to the new address immediately after the mutable borrow returns. Upon identifying the owner's address change, each immutable borrow would direct a server to fetch a fresh version of the object from the new address as opposed to relying on a stale copy residing in its cache.

Note that this is a general protocol that covers the case that the object is on the same server that issues the write—as long as the server moves the object into a different location in the global heap, no other servers can read the stale copies of the object. However, this is not efficient as each local write requires moving the object to a new address. To address this inefficiency, DRust employs a pointer-coloring technique, inspired by the designs of many

[1]The terms "copy" and "move" are used to describe the processes of adding an object into a cache without changing its global address, and relocating the object into a server's heap partition, which requires changing its global address.

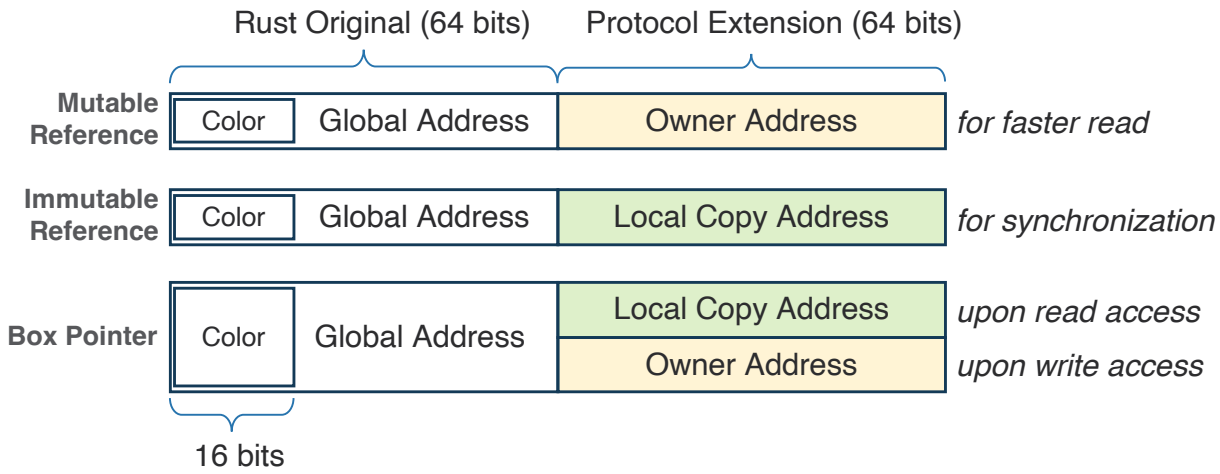


Figure 5.3: DRust repurposes Rust pointers and references to contain a global heap address and an extension field for its coherence protocol.

concurrent garbage collectors [2, 77]. Discussed at the end of this subsection, this technique offers a more efficient solution for handling local writes.

Pointer Layout. In order to support this protocol, each pointer must remember not only the object's global address, but also the address of the cached copy in a server's local cache (to avoid redundant remote fetches). As such, we modify Rust's pointer structure, as illustrated in Figure 5.3. DRust internally extends each Rust `Box` pointer and reference with an additional 64-bit field, which is used differently for read and write access. At a high level, the field records the address of the cached copy for faster read accesses; for write accesses, this field records the address of the object's owner for post-write synchronization. Additionally, DRust reserves the highest 16 bits in the global address field as "color" bits. These bits record the version number of the pointer and play a crucial role in DRust's efficient handling of local writes.

Next, we discuss how DRust reimplements Rust's ownership operations to realize the distributed coherence protocol.

Algorithm 8: Access logic for mutable references.

Input: A mutable reference m containing a global address $m.g$ and the owner address $m.o$.
Output: A local memory address to be written to.

```
1 Function DerefMut(m):
2     if ¬IsLocal(m.g) then
3         m.g ← Move(ClearColor(m.g))
4     return ClearColor(m.g)
5 Function DropMutRef(m):
6     c' ← GetColor(m.g) +1
7     Write(m.o, AppendColor(m.g, c'))
```

Mutable Borrow. Mutable borrow creates a mutable reference that holds exclusive access to the referenced object for writing. Algorithm 8 outlines the procedures for both dereferencing and dropping a mutable reference. When performing dereferencing, DRust first checks the object's location (Line 2) and performs direct access if the object's address belongs to the heap partition of the machine A that executes the access. Otherwise, DRust *moves* it to A's heap partition (as opposed to caching it) (Line 3). The move, conducted in the following three steps, changes the object's global address. DRust (1) copies the object into A's heap at an address p, (2) updates the mutable reference with the address p, and (3) asynchronously requests the remote server that previously stored the object to deallocate the original object.

A challenge arises with its original owner `Box`, which now becomes a dangling pointer, pointing to an invalid memory location. Fortunately, the integrity of the system is maintained by the single-writer invariant (referenced as Invariant 3). This invariant ensures that while the mutable reference remains alive, no other entity, including the original owner, can access the data. To ensure correctness, when this new reference is dropped, DRust synchronously updates the original owner `Box`, redirecting it to the new address p (Line 7). As a result, the original owner always possesses the latest view of the object. Additionally, all modifications made through this mutable reference are visible in all subsequent accesses, as they necessitate borrowing permission from the updated owner `Box`. The single-writer invariant also eliminates the possibility of simultaneous updates to the owner, ensuring that updating the owner is

Algorithm 9: Access logic for immutable reference.

Input: A shared immutable reference r containing a global address $r.g$ and a local copy address $r.l$, and a local cache hashmap H.

Output: A local memory address for reading.

```
1  Function DEREF(r, H):
2      if ISLOCAL(r.g) then
3          return CLEARCOLOR(r.g)
4      else
5          if r.l = Null then
6              Atomic {
7              if r.g ∈ H then
8                  ⟨l', cnt⟩ ← GETENTRY(H, r.g)
9                  r.l ← l'
10                 UPDATEENTRY(H, r.g, ⟨l', cnt + 1⟩)
11             else
12                 r.l ← COPY(CLEARCOLOR(r.g))
13                 INSERTENTRY(H, r.g, ⟨r.l, 1⟩)
14             }
15         return r.l
16 Function DROPREF(r, H):
17     if r.l ≠ Null then
18         Atomic {
19         ⟨l', cnt⟩ ← GETENTRY(H, r.g)
20         UPDATEENTRY(H, r.g, ⟨l', cnt − 1⟩)
21         }
```

free from concurrency issues.

Immutable Borrow. Immutable borrowing allows concurrent reads to the same object from immutable references on the same or different servers. As detailed in Algorithm 9, DRust handles the dereferencing of immutable references by first checking the object's location (Line 2). For remote objects, DRust creates a local copy in the *per-node read-only "cache"* and records its local address in the reference's extension field (see Figure 5.3). This preserves the original global address of the object, ensuring that any new immutable reference—whether it is derived from the owner `Box` or from another immutable reference—can always access the

original object from the global heap.

As opposed to being a separate memory space, our "cache" provides a "virtual" aggregation of all local copies maintained on each server. These copies reside in the regular heap, managed by a per-node hashmap H. This hashmap maps each global address to a pair of its local address and the number of local immutable references to the local copy. To prevent redundant copies of an object on the same server, DRust checks the hashmap H before creating a new local copy (Line 7). If a local copy is already present, DRust increments its reference count in H and updates the extension field in the immutable reference to point to this copy (Lines 8–10). If no existing copy is found, a new one is created (Lines 12–13). Since the hashmap uses objects' global addresses as keys, if an object has been modified by another server since its last read, its global address must have changed, making cache lookup fail even if a (stale) local copy exists.

DRust actively updates the reference count of each local copy when an immutable reference is either dereferenced or dropped, as outlined in Lines 10 and 20. Utilizing these counts, the DRust runtime periodically scans the "cache" and lazily reclaims unreferenced copies (*i.e.*, those with a zero reference count) under memory pressure (§5.3.2.1). This mechanism, in conjunction with the safe borrowing invariant (2), prevents the local cache from memory leaks or illegal accesses.

Owner Access without Borrow. DRust treats a direct memory access via the owner `Box` as a pair of mutable/immutable borrow and return. Depending on the reference type, DRust uses the extension field of `Box` accordingly and executes the read/write dereferencing logic. A special case arises when a mutable owner is immutably borrowed and becomes immutable until all borrowed references return. In this case, the owner can only cache the object during the borrow and delay the move until the borrow finishes. This would not create any correctness issues because the owner cannot be used for write access during this period.

Ownership Transfer. Similar to Rust, DRust does not move the actual value during the transfer and only copies the `Box` pointer. DRust additionally checks and resets the pointer's extension field and frees the cached copy in the executing machine's cache to avoid cache leakage.

Memory Deallocation. Like Rust, DRust tracks the lifetime of an object via its owner. Given that ownership transfer is implemented by only evicting the cached copy of the object (without changing its global presence), the memory safety of DRust's global heap is preserved by the singular owner invariant (1). In other words, DRust still guarantees that when an object's owner goes out of scope, the object must be unreachable (and dead) and can be safely deallocated.

Consistency Model. Our protocol, together with Rust's ownership model, offers *sequential consistency* for cross-server memory accesses in safe Rust programs (*i.e.*, following the original Rust, no guarantees can be provided when Rust `Unsafe` is used), which is a strong consistency order. Therefore, it allows any safe Rust program to preserve its memory consistency on DSM. Sequential consistency necessitates a coherent memory system, requiring not only the SWMR invariant but also the *data-value* invariant [85]. In simple terms, the data-value invariant requires that the latest write to a value is immediately visible to subsequent readers. As discussed earlier, DRust's protocol moves an object upon a write and updates the owner immediately. Therefore, the latest value is globally visible after each mutable borrow finishes. Subsequent read accesses, either in the Owned state or the Shared state, are hence guaranteed to see the moved object and read its latest value.

Optimizing for Local Writes. A special case is that a server issues a write to an object that resides in its own heap partition. While the coherence protocol still guarantees safety, requiring moving an object in its local heap each time it is written clearly brings inefficiencies. To optimize for local writes, DRust adopts a pointer-coloring method, inspired by the design

Algorithm 10: Utility functions for pointer coloring.

```
1 Function GetColor(g):
2     return g ≫ 48
3 Function ClearColor(g):
4     return g & ((1 ≪ 48) − 1)
5 Function AppendColor(g, c):
6     return ClearColor(g) | (c ≪ 48)
```

of concurrent garbage collectors in a managed runtime system such as JVM [2, 77]. Several utility pointer coloring functions are shown in Algorithm 10 which are used when dereferencing and dropping a reference. We reserve the first 16 bits of a global address as a "color". The color value stored in the object's owner gets incremented upon the expiration of a mutable reference, as detailed in Lines 6–7 in Algorithm 8. Any subsequent immutable borrow would look up the cache with the object's global address. Even if the actual address remains the same, its color changes if a write has occurred. As such, the lookup would not return any stale copy from the local cache.

The 16-bit `color` field may overflow when the pointer keeps being borrowed for local writes on the same server. DRust implements a *move-on-overflow* strategy that moves the object to a new address and resets its color to zero once the maximum color value is reached (2^{16}), thereby preventing overflow and maintaining system integrity and performance.

Writing Unsafe Code in DRust. Rust allows developers to bypass compiler safety checks and write *unsafe* code for low-level operations such as accessing raw pointers and mutating shared variables at their own risk [66, 103]. Since DRust relies on SWMR semantics enforced by Rust's ownership types, DRust ensures consistency and memory safety only in the "safe" Rust code. DRust does not cache objects in unsafe code but allows developers to implement their own cache. Developers must ensure that they do not violate consistency in unsafe code blocks where type safety is not enforced. This caution mirrors practices in other managed languages, like native code in Java and unsafe code in C#. To assist developers, DRust offers

primitives such as `dalloc`, `dread`, and `dwrite` for managing data on the global DSM heap.

5.3.1.2 Adapting Rust Standard Libraries

To further reduce the barrier for programs to run distributively, we reimplement several standard Rust libraries atop DRust's core memory constructs covering four categories: threading for distributed computation (`std::thread`), inter-thread channel for communication (`std::sync::mpsc`), reference-counted pointers for ownership sharing (`std::sync::Rc` and `std::sync::Arc`), and shared-state locks for concurrency control (`std::sync::Mutex` and `std::sync::atomic`).

Threading. DRust's threading library enables Rust threads to run distributively with two major adaptations. First, it enables distributed thread launching by re-implementing the `spawn` interface. Internally, it captures the thread body as a closure during compile time and forwards it to the runtime. During execution, the runtime launches the thread according to each server's load (details in §5.3.2.1). Second, DRust performs implicit ownership transfers between the parent and the child threads at the start or the end of the child thread execution. Thanks to the distributed ownership transfer support provided by DRust's memory model, the implementation in the threading library is hidden from developers and preserves type soundness and memory safety. Additionally, DRust is compatible with advanced thread utilities such as `thread::scope`, which allows for the spawning of scoped threads that can borrow non-static data. These utilities ensure that all threads are joined at the end of their scope and can internally utilize DRust's functions for spawning and joining threads, thus extending their applicability to the distributed setting.

Inter-Thread Channel. DRust extends Rust's channel to connect two distributed threads for message passing. DRust internally builds a network-based message queue for cross-server messages. Benefiting from the shared global heap, `Box` pointers and references can be safely

copied and remain valid across servers. Therefore, the sender can push an object into the channel *as is* without serialization, even if it may contain `Box` pointers. DRust forwards the object binary bytes to the receiver over the network, and the receiver can recover the object from the binary by direct type conversion without deserialization.

Ownership Sharing. Rust allows multiple owners to share an object via reference-counted smart pointers, which count the number of live owners. In this case, smart pointers only have read access, and the object lifetime terminates when all owners die and the reference count hits zero. DRust does not require special treatment for `Rc` as it only allows ownership sharing inside a single thread. For `Arc` which shares ownership among multiple threads, DRust handles it in a similar way to immutable references with on-demand local caching and lazy eviction.

Shared-State Concurrency. Rust supports shared-state concurrency, primarily through its atomics and mutexes, where threads commonly share an atomic-typed value or one mutex via ownership sharing (*i.e.*, `Arc`). Unfortunately, the ownership model cannot type check concurrent read/write to shared states. Hence, Rust relies on an *unsafe* implementation in its standard library. §5.3.1.1 already provides a general discussion on writing unsafe code in DRust, and here we focus on DRust's implementation for distributed shared states.

Shared states create a unique challenge for DRust, as they may be replicated on multiple servers and those states must be synchronized among these servers. For example, an `Arc<AtomicBool>` may be replicated across different servers and used independently, causing multi-version issues if not synchronized properly. DRust addresses this inconsistency by allocating the actual value on the global heap and storing only the `Box` pointer in `atomic` types. This design allows atomics to be freely moved or replicated across servers while keeping a single version of the actual value. To synchronize concurrent operations on atomics, DRust adapts methods of atomic types to forward the operation as a message to the server storing the actual value, which serializes all operations with unsafe logic similar to the original Rust

```
pub struct Node { val: i32, next: Option<TBox<Node>>, }
pub struct List { pub head: Option<Box<Node>>, }
impl List {
  pub fn sum(&self) -> i32 {
    let mut total: i32 = 0;
    if let Some(r) = &self.head {
      let mut node = &**r; // Fetch whole list to local.
      loop { // Iterate every list node.
        // Accessing node is guaranteed local.
        total += (*node).val;
        if let Some(next) = &node.next {
          node = &**next;
        } else { break; }
      }
    }
    total
  }
}
```

Listing 5.2: A linked list implementation with `TBox` in DRust. The use of `TBox` ties list nodes one by one. Iterating a list will fetch all nodes together (if they are on another server), and henceforth accessing any node is guaranteed local.

to guarantee atomicity and memory consistency. Similarly, DRust implements `Mutex` by allocating its metadata and owned object on the global heap and leaving only `Box` pointers in the mutex struct. Concurrent operations on mutexes are serialized on the server storing the mutex.

5.3.1.3 Affinity Annotations

To further improve performance, DRust allows developers to provide optional data affinity semantics via annotations. These annotations are useful for many datacenter applications that make extensive use of object-oriented data structures that require *pointer-chasing* to access. For instance, Memcached [6] uses a chained hash table where each hash bucket stores its KV pairs with a linked list. To find one KV pair from a bucket, Memcached has to iterate the linked list following the node pointers. However, frequent pointer chasing is unfavorable in a distributed setting, where each pointer dereference incurs additional runtime checks and potential cross-server traffic. It would be beneficial for the runtime to colocate them on the same server and schedule the computation there.

```
1 fn main() {
2   let val: Box<i32> = Box::new(5);
3   let mut a = Accumulator{val};
4   let remote_add = spawn_to(a.val, move ||
5     a.add(10)).join(); // a.val == 15
6 }
```

Listing 5.3: A distributed accumulator can leverage `spawn_to` to offload a thread to the server where `a.val` locates.

Data-Affinity Pointer. To expose data affinity for clustered placement, DRust includes a new pointer type `TBox` for developers to "tie" a heap object to its owner. `TBox` shares the same interfaces as the ordinary `Box` and can be used as a drop-in replacement for `Box`. However, `TBox` enforces that the pointed-to object must reside on the same server as its owner. In other words, when its owner object is copied or moved, the object referenced by `TBox` will be copied or moved as well. `TBox` can be used in a nested manner to allow a large union of objects to be co-located. The DRust runtime fetches (*i.e.*, copies or moves) them together in a single batch, leading to fewer network round-trips and higher throughput. `TBox` can also be assigned to and owned by a stack variable, in which case the referenced object is pinned onto the heap partition of the server that hosts the stack and cannot be moved. Dereferencing a `TBox` is thus guaranteed to be a local access—DRust optimizes it by skipping the runtime check.

Listing 5.2 presents a linked list implementation using `TBox`. Our linked list uses `TBox` (Line 1) to specify the data affinity between consecutive nodes. As a result, all list nodes are chained with `TBox`, forming an affinity group. Line 4–17 define a `sum` function that calculates the total sum of all node values. Assuming the list is non-empty, Line 7 dereferences the pointer to the head node, and the DRust runtime checks the location of the head node and fetches the entire list of nodes together if they are not local. Next, accessing each node inside the loop body (Line 8–14) is guaranteed local and hence skips runtime checks. Compared to using `Box` directly, `TBox` makes both data fetching and accessing more efficient.

Data-Affinity Thread. To expose the affinity between data and computation for thread scheduling, DRust extends its threading library with a `spawn_to` interface. `spawn_to` mirrors

the ordinary `spawn` interface to spawn a new thread but takes an additional `Box` pointer argument, which indicates where the thread should be created. The runtime checks where the `Box` points to and creates the new thread on that same server. A common practice to use `spawn_to` is to pass the mostly-accessed object as the location indicator. Listing 5.3 presents how the distributed accumulator (shown in Listing 5.1) can use `spawn_to` to offload a thread to the same server as `a.val` resides. Line 5 hence performs local dereference to `a.val` inside `a.add()`.

5.3.2 DRust Runtime System

DRust's runtime system is the core component that manages memory and compute resources. It includes a runtime library (§5.3.2.1) that is linked to each application and launched on each server and a cluster-wise global controller (§5.3.2.2).

5.3.2.1 Application-Integrated Runtime

The runtime library consists of a communication layer to support inter-server coordination and data transfer, a heap allocator to manage the heap partition and the read-only cache, and a thread scheduler to launch and schedule application threads.

Communication Layer. The DRust runtime uses its communication layer to support the cache coherence protocol and cross-server memory accesses. The communication layer consists of a control plane and a data plane, both built with RDMA. The control plane leverages *two-sided verbs* to send and receive small control messages, and the receiver can perform the coherence logic upon receiving the message to minimize the end-to-end latency. The data plane, in contrast, is specialized for bulky data transfer with one-sided verbs. It fetches an object as a whole with a single RDMA message upon pointer dereferencing without interrupting the target server, minimizing both latency and CPU usage.

Heap Allocator. The DRust runtime provides standard memory allocation interfaces and always returns global addresses to the upper-level language abstractions. It prioritizes local memory allocation as long as the local heap partition has sufficient space. This strategy improves data locality by colocating an object with its creating thread.

When the local heap partition is depleted, DRust queries the global controller and allocates memory on the most vacant server. For remote memory allocation, it forwards the request to the target server by sending a message through the communication layer and returns the allocated address to the user. Memory deallocation follows a similar logic but it bypasses the controller and finds the server directly via the object's global address. The allocator does not reserve separate space for the local cache. Instead, it manages the cache as part of the local heap partition and always allocates cached entries in the local heap partition. Under memory pressure, the allocator will scan the local cache and evict entries that are no longer used (*i.e.*, reference count hits zero).

Thread Scheduler. The DRust thread scheduler runs in the user space and schedules threads locally to maximize CPU utilization. It also provides thread migration functionalities, facilitating the global controller to balance load between busy and vacant servers.

The scheduler represents a newly created user thread as a closure, which includes a function pointer and a set of initial arguments (*i.e.*, references). It collaborates with the global controller to allocate a unique stack space for a thread (see Figure 5.2), and starts the thread by executing the closure.

The scheduler adopts the method of cooperative multitasking and context switches between threads *non-preemptively*. A running thread yields its control flow proactively when developers call `await` or reactively upon long-latency operations. Similarly to other systems [88, 96, 121], our scheduler handles context switches as function calls, which allow DRust to save only a few registers per thread.

The scheduler supports creating/migrating a thread to another server as well. To migrate

a thread, DRust sends its function pointer, the saved register state, and its stack to the target server. Because each thread reserves its stack address range globally, DRust can copy the stack across servers without changing its address. Therefore, the thread can be easily resumed by directly calling the function pointer with the saved register state on the target server. DRust generates code for state transmission during the compile time for the scheduler to call upon thread migration.

5.3.2.2 Global Controller

The controller runs as a daemon process on the machine where the program is launched. It manages cluster resources and coordinates memory allocation and thread migration. It periodically pings each server to probe and record its resource usage (CPU and memory). It controls resource allocation in cooperation with the DRust runtime on each server. When allocating memory or creating a thread, the runtime will first query the controller, which chooses a target server following adaptive policies (discussed later), and then coordinate with the runtime on the target server to perform the actual operation. The controller also maintains a global table to track the location of each thread; the table is queried and updated by the scheduler when migrating a thread.

During program execution, servers may run into imbalanced loads when objects get relocated or new threads are created. DRust balances the load of each server by *migrating* threads from the busy server to less occupied ones, following an adaptive policy to minimize cross-server memory accesses. If a server is about to run out of memory (>90% memory usage), the controller keeps migrating the thread that consumes the most local heap memory until the pressure is resolved. If the server is under compute congestion (>90% CPU usage), the controller migrates threads that frequently access remote objects. The thread is then moved to the server it accesses the most unless the target server is also overloaded, in which case it moves to a vacant server instead.

5.3.2.3 Fault Tolerance

In DRust, the global heap can be replicated to tolerate failures. Replication creates copies for each heap partition at the same virtual address on backup servers. Threads, in contrast, are not replicated for efficiency and are only executed with the primary global heap. To maintain a synchronized view between the primary heap partition and its backup copy, a thread must additionally write back to the backup partition after each mutable borrow. However, our insight is that the thread can batch modifications to an object and delay the write-back until the object ownership is transferred to another server, which is the time point that the object becomes visible to threads on other servers. When a server with a primary heap partition fails, the controller will automatically promote its backup server to the primary and add a new backup server.

5.4 Implementation

The majority of DRust was implemented in Rust except for its communication library which is in C. We implemented DRust's core language constructs as three Rust types (*i.e.*, `struct`): `Ref<T>`, `MutRef<T>`, and `DBox<T>`. They serve as the counterpart for the original Rust `&T`, `&mut T`, and `Box<T>`, respectively. We implemented the coherence protocol with traits on these types, including `Copy`, `Clone`, and `Drop`, which are automatically embedded into the program source code and executed when references/pointers are created or destroyed. To support unmodified Rust programs, we changed the Rust compiler and added additional compilation passes to transform Rust references and `Box` pointers into corresponding types in DRust.

Our communication layer links `libibverbs` directly for fast and kernel-bypassing RDMA networking. We implemented a low-level C library that covers basic connection establishment and exposes high-level Rust interfaces for various RDMA verbs, including `RDMA_READ`, `RDMA_WRITE`, `RDMA_SEND`, `RDMA_RECV`, `ATOMIC_FETCH_AND_ADD`, and `ATOMIC_CMP_AND_SWP`. We primarily utilize one-sided `READ` and `WRITE` verbs for data transfers between servers, as they

outperform the two-sided `SEND/RECV` counterparts—one-sided operations bypass the CPU and OS at the receiver side, whereas two-sided operations require the receiver to pre-post `RECV` verbs and await notification upon message arrival. For instance, when a remote object is accessed via mutable references, DRust copies the object to local memory using the `READ` verb. Upon dropping the reference, DRust updates the original owner `Box` to reflect the new address, a process executed using the `WRITE` verb. Conversely, two-sided `SEND/RECV` verbs are utilized for control message exchanges, such as establishing connections across servers. Atomic verbs `ATOMIC_FETCH_AND_ADD`, and `ATOMIC_CMP_AND_SWP` are primarily utilized for implementing shared states (e.g., atomic types and mutexes). DRust uses the RC (reliable connection) transport type to ensure reliable transmission and strict message ordering.

Our heap allocator implementation piggybacks Rust's original allocator and aligns its virtual address range with the heap partition range. Our thread scheduler was built upon Tokio [121] for its efficient user thread and cooperative scheduling integration. The global controller is responsible for managing all threads in the cluster and padding their stacks to avoid address overlapping.

5.5 Limitations

DRust's design has three limitations. First, although DRust permits the use of unsafe code, its consistency guarantees are only applicable to safe Rust code. In unsafe code blocks, developers are responsible for ensuring consistency themselves. Second, DRust's superior performance relies on SWMR semantics exposed by applications. In cases where data is mostly under shared states (such as `Mutex`), DRust degenerates into a traditional DSM system; all concurrent accesses to the same data have to be centralized and serialized by the server responsible for the shared states. However, such scenarios contradict Rust's recommended programming practices. Finally, the current implementation of DRust does not support address space layout randomization (ASLR) yet, and we have temporarily disabled it.

However, DRust's design is compatible with ASLR as long as DRust threads share the same randomized address space layout on each server, which will be supported in future versions of DRust.

5.6 Evaluation

Setup. We evaluated our system on an 8-node cluster, where each node was equipped with dual Intel Xeon E5-2640 v3 processors (16 cores), 128GB of RAM, and a 40 Gbps Mellanox ConnectX-3 InfiniBand network adapter, connected by a Mellanox 100 Gbps InfiniBand switch. All servers ran Ubuntu 18.04 with kernel 5.14. We disabled hyperthreading, CPU frequency scaling, OS security mitigations in accordance with common practices [101, 107].

Methodology. We compared DRust with two state-of-the-art DSM systems, GAM [27] and Grappa [88]. For a fair comparison, we ported the evaluated applications to each baseline system and invested extensive effort in tuning parameters to achieve their best possible performance. GAM offers ordinary object read/write interfaces, and we exported it as a library to Rust and hooked pointer dereferencing to use GAM's API without program modification. Grappa, in contrast, offers a drastically different programming abstraction that requires rewriting the program to access shared memory via *delegation*. Therefore, we re-implemented applications in C++ and re-structured them using Grappa's abstractions.

5.6.1 Applications

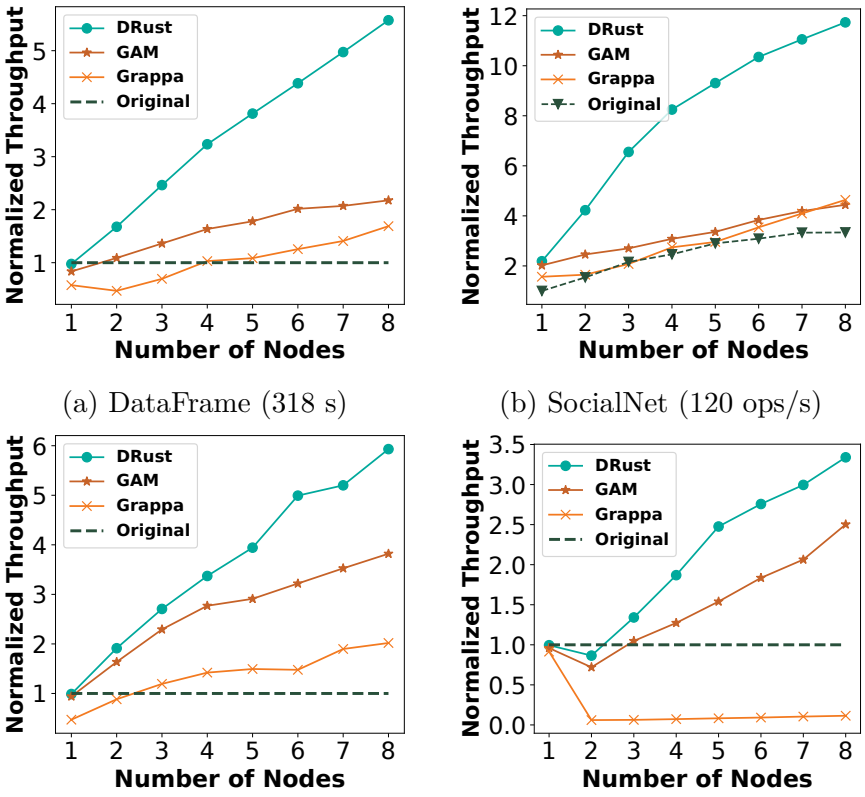
We evaluated four representative datacenter applications covering a wide range of use cases and resource demands, including data analytics, microservices, scientific computation, and key-value storage, as shown in Table 5.1.

Application	Dataset	Memory (GB)	Comp. Intensity (cycles/byte)
DataFrame [99]	h2oai [55]	64	110.13
SocialNet [48]	Socfb-Penn94 [104]	64	86.09
GEMM [23]	LAPACK[10]	96	300.63
KV Store [27]	YCSB [38]	48	48.15

Table 5.1: Applications used in the evaluation.

DataFrame is an in-memory data analytics framework similar to Spark [135] and Pandas [130]. We built our library atop Polars [99], a native DataFrame engine in Rust offering OLAP query APIs such as `filter`, `groupby`, and `join`. DataFrame organizes the dataset as columnar format tables in shared memory, and user queries will manipulate table columns by reading/writing rows and transforming them into new tables. DataFrame exploits data-level parallelism by internally partitioning columns by row into an array of small chunks where each chunk can be processed independently. We additionally applied `TBox` to annotate chunks from the same table column for co-location and used `spawn_to` to offload columnar operations to the data side to improve data locality and performance. Note that such annotations were not necessary for the application to run; they were added for additional performance optimizations.

SocialNet is a twitter-like web service from the DeathStarBench suite [48]. It is composed of 12 microservices with complicated call dependencies. Each microservice in SocialNet can scale independently with replicas, thereby offering higher throughput with more servers. SocialNet decouples the process of user texts, media resources, and storage into different microservices, and it employs RPCs to pass values (texts, media files, *etc.*) between them. DRust enables SocialNet to pass only references in RPCs, eliminating the serialization/deserialization overhead and redundant data copies. Because SocialNet was implemented in C++ and deployed with Docker Swarm [43], we ported it into Rust for our evaluation. We followed its original microservice structure but changed the RPC call sites to pass references instead



(a) DataFrame (318 s)

(b) SocialNet (120 ops/s)

(c) GEMM (1039 s)

(d) KV Store (2.7 Mops/s)

Figure 5.4: Application throughput when running with DRust, GAM, and Grappa, normalized to the throughput of their original implementation running on a single node. The number in the parenthesis is the original application's throughput on a single node.

of values, and we followed the original orchestration configuration to spread and scale each microservice in the cluster. We did not use any affinity annotations for SocialNet.

GEMM (general purpose matrix multiplication) is a highly-optimized matrix multiplication routine from the BLAS library [23]. We ported the library using the same divide-and-conquer algorithm by recursively partitioning each matrix into small chunks for parallel processing and reducing the final results. Input and output matrices are stored in the shared memory, where each subroutine will read two input matrix chunks and write the partial results back to the output matrix. Our port strictly followed the original implementation without using additional affinity annotation.

KV Store is an in-memory key-value cache engine similar to Memcached [6]. It uses a hash table to store KV pairs in shared memory and mutexes to synchronize concurrent requests. We used YCSB benchmark [38] to generate zipf load with 90% GET and 10% SET using default skewness parameter 0.99.

5.6.2 Scaling Performance

In this experiment, we investigated whether DRust can speed up applications by distributing them in a cluster and how well they can scale with the number of servers used. For each application, we first ran it *as is* on a single server without using DSM and measured its throughput. Then, we ran the same application on DSM (subject to modifications when running Grappa) with the same configuration but on varying numbers of servers and measured the throughput normalized to its single-node throughput (*i.e.*, strong scaling). As GAM and Grappa cannot adaptively balance the workload across servers, we evenly distributed the application's working set and threads among all participating nodes. Ideally, an application should scale linearly and enjoy proportionally higher throughput with more nodes. However, this is usually unachievable because of the limited parallelism of real-world applications and the coherence overhead of DSM systems, and a good result for DRust will show that applications' throughput can get close to their ideal throughput.

Figure 5.4 shows the results for each application respectively. DRust outperforms both GAM and Grappa in all cases. On a single node, it is 1.04–2.10× faster than two baseline DSMs, while only adding a maximum overhead of 2.42% compared to the original program. When running with multiple nodes, DRust scales up applications significantly better than GAM and Grappa. On eight nodes, DRust achieves a throughput that is 1.33–2.64× higher than that of GAM, 2.53–29.16× higher than that of Grappa.

Compared to each program's single-machine performance, using DSM over DRust enables each program to easily leverage the available distributed resources and achieve a throughput that is 3.34–11.73× higher than their single-machine counterparts. Next, we discuss each

application to explain the scalability difference between DRust and the baseline DSMs.

DataFrame. As shown in Figure 5.4a, compared with its original version, DataFrame running on eight nodes with DRust achieves 5.57× higher throughput, whereas with GAM and Grappa, the throughput improvements are 2.18× and 1.69×, respectively. In other words, DataFrame with DRust is 2.56× and 3.29× faster than GAM and Grappa on eight nodes, respectively.

A detailed examination reveals that the performance difference comes from the *shared index table* in each DataFrame operation and the *shared chunks* between dependent DataFrame operations. In each operation, DataFrame constructs an index hash table to track the mapping from each destination chunk in the output column to all its source chunks in the input column. This index table is shared by all index-builder threads and worker threads. During processing, index-builder threads will concurrently insert into the index table using the destination chunk ID as the key and an array of source chunk IDs as the value, and worker threads will look up the shared index table and retrieve source chunks for processing. As a result, the massive writes and reads to the shared table can incur heavy coherence overhead. Further, DataFrame passes chunks as references between dependent operations and relies on the DSM system for actual data movement. However, it only performs lightweight computation over the fetched data (*i.e.*, low compute intensity as shown in Table 5.1), making the coherence overhead stand out.

DRust outperforms GAM and scales much better because of its light coherence protocol, which incurs negligible object move overhead for writes and no coherence overhead for reads. The use of affinity annotations also helps DataFrame colocate worker threads with their frequently accessed data, bringing 20% additional boost (details in §5.6.3). GAM, in contrast, has to invalidate each cache block upon each write and read, thereby bottlenecked by the extensive coherence traffic. Grappa performs the worst in all three DSM systems due to its *always-delegation* programming model, which implements every global memory read/write

via a delegated function call. The cost for delegation overwhelms the actual memory access latency in this case, ruining the performance of the shared hash table. Grappa's delegation overhead actually causes a 1.23× slowdown when scaling DataFrame from a single node to two nodes.

SocialNet. Since SocialNet is microservice-based and can be deployed distributively, we added another baseline by running the original (non-DSM) code but deploying it on varying numbers of nodes. Figure 5.4b demonstrates the performance of all systems. SocialNet runs consistently faster with all three DSM systems compared to the original version. DRust, GAM, and Grappa achieve a 2.18×, 2.02×, and 1.57× speedup on a single node and a 3.51×, 1.33×, and 1.39× speedup on eight nodes, respectively. In the conventional setup, SocialNet requires data—such as text and media files—to be serialized into byte streams for network transmission, and then deserialized back into usable formats at the receiving end. This serialization and deserialization process is computationally intensive, particularly for large or complex data objects. In contrast, DSM systems enable SocialNet to pass references instead of the entire data values required by remote procedure calls. This approach eliminates the need for serialization and deserialization, reduces redundant data copies, and significantly enhances performance. DRust scales much better than GAM and Grappa thanks to its lightweight coherence protocol, achieving up to 2.77× and 3.16× higher throughput than GAM and Grappa, respectively.

GEMM. GEMM differs from the previous two applications in its high compute intensity and relatively infrequent shared memory accesses. In this application, matrices are transformed and divided into smaller sub-matrices for parallel processing. Each computing thread, responsible for multiplying sub-matrices, is assigned to a server. These threads cache their respective sub-matrices in the server's local memory and access them repeatedly to compute product results. This process is highly compute-intensive. As depicted in Figure 5.4c, DRust and GAM scale well for GEMM and achieve 5.93×, 3.82× speedup with eight nodes. In

contrast, Grappa only achieves a 2.02× speedup with eight nodes due to its inability to cache sub-matrices locally, necessitating frequent remote accesses. DRust's superior performance over GAM, with a 1.55× higher speedup on eight nodes, is primarily due to its more efficient handling of initial cross-server data accesses required when a sub-matrix is first accessed remotely. Unlike GAM, which incurs significant runtime overhead due to the maintenance of state and location of data copies, DRust directly copies data to local memory, without any complex cross-server synchronization operations, thus enhancing overall efficiency.

KV Store. KV Store is the most DSM-unfriendly application in our evaluation because it exposes poor memory locality and low compute intensity, which amplifies the overhead of cross-server memory accesses. In addition, it uses mutexes to synchronize between workers and the structure of the program does not lend itself to ownership-based read/write ordering.

Figure 5.4d shows the results. KV Store experiences a slowdown on all three DSM systems when scaling from a single node to two nodes (13% for DRust, 25% for GAM, and 93% for Grappa). However, the impact is mitigated when more servers are enlisted—DRust and GAM achieve 3.34× and 2.50× higher throughput on eight nodes compared to the original KV Store implementation, respectively. Due to the limited ownership semantics exposed by mutexes, DRust does not scale as well with KV Store as with other applications. DRust is 1.33× faster than GAM on eight nodes, benefiting from its adaptive load balancing and a more efficient implementation of mutexes utilizing one-sided RDMA atomic verbs, whereas GAM depends on less efficient two-sided RDMA messages for synchronization. Grappa, in contrast, incurs the highest distribution overhead and poorest scalability, primarily because each PUT/GET operation requires remote delegation, and nodes handling popular objects become bottlenecked due to skewed load.

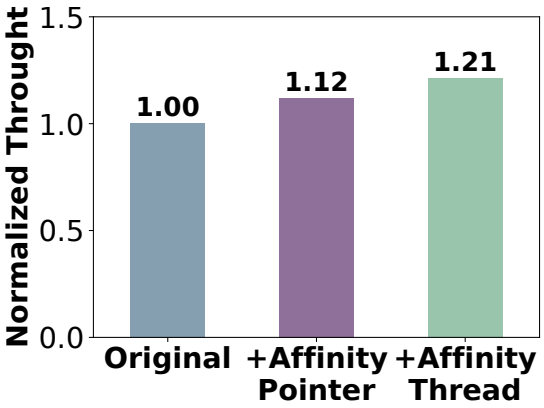


Figure 5.5: Effectiveness of DRust's affinity annotations.

Latency (cycles)	Average	Median	P90
`DRust`	395	356	536
`Rust`	364	332	496

Table 5.2: DRust's `Box` pointer only adds a small dereferencing cost compared to Rust's ordinary `Box`.

5.6.3 Drill-Down Experiments

Affinity Annotations. In this experiment, we evaluated the individual contributions of affinity annotations by enabling each of them incrementally for DataFrame on eight nodes. Figure 5.5 reports the results. Using `TBox` helps DataFrame group chunks from the same column and eliminates the runtime dereference check overhead for single-column operations (*e.g.*, `filter`), bringing a 12% throughput improvement. Adding `spawn_to` further improves the throughput by 9% by informing DRust runtime to colocate the worker thread to its input columns, which reduces cross-server memory accesses.

Runtime Dereference Checks. We measured the latency of dereferencing DRust's `Box` pointer and compared it with an ordinary Rust `Box` pointer. Both of them point to an 8-byte object in local memory and not in CPU's cache, which represents the common path for pointer dereferencing. Table 5.2 reports the results. DRust only adds a small overhead of ~30 cycles. Note that this microbenchmark is extremely memory-intensive, whereas real-world applications usually employ larger object sizes and are more compute-intensive, further mitigating the runtime check overhead. For our evaluated applications, we observed a 1.02% overhead for DataFrame and a 1.14% overhead for BLAS, when they run with DRust on a single node, respectively.

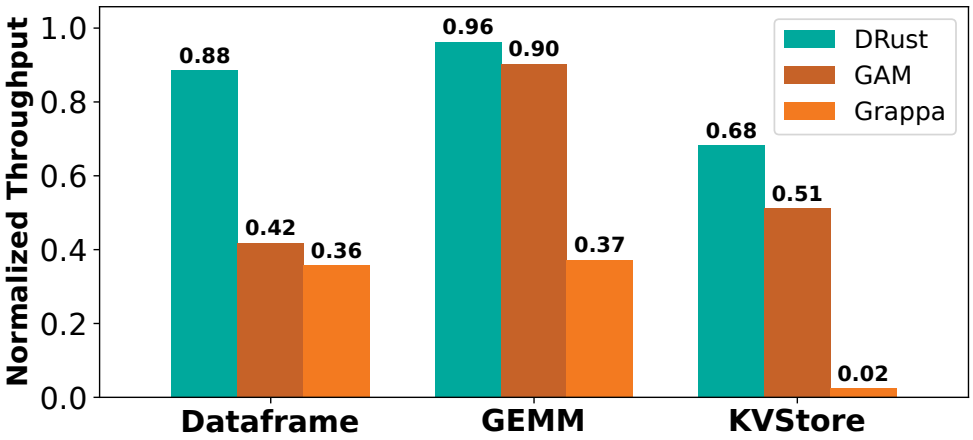


Figure 5.6: Comparison of cache coherence costs between DRust, GAM, and Grappa on eight nodes.

Thread Migration Latency. To quantify how quickly DRust can resolve the workload imbalance, we measured the latency for the DRust runtime to migrate a thread by running GEMM on eight nodes and repeated the experiment for ten times. On average, DRust migrated 15 threads with an average of 218µs latency for each migration.

Cost of Cache Coherence. In this experiment, we ran each application again on a single node and eight nodes but fixed the total amount of CPU and memory resources. For the eight-node setting, we distributed the resources evenly to each node and measured application throughput. We expect to see a slowdown due to the cost of running the coherence protocol and cross-server memory accesses, but a good result for DRust should show that application performance remains close to its original single-node version. Figure 5.6 reports the results. SocialNet uses pass-by-value RPCs in its original version and is significantly slower than our DSM-based version, so it is omitted in the evaluation. DRust adds only moderate cache coherence cost with an overhead of 32% in the worst case (KV Store) and 4% in the best case (GEMM). GAM and Grappa, in contrast, incur much higher overheads ranging from 10% to 98% for different applications.

5.7 Related Work

Software DSM Systems. Distributed cache coherence protocols and their implementations for DSM have been extensively studied since 1980s [30–32, 46, 54, 74, 75, 83, 91–93, 118]. Among them, Munin [21] and TreadMarks [16] proposed relaxed consistency models and simpler protocols trying to alleviate the coherence overhead. Recent DSM systems leveraged today's advanced hardware such as RDMA [27, 68, 88, 112, 119, 136] to improve efficiency.

Disaggregated and Remote Memory. Memory disaggregation and remote memory techniques are another promising approach to scaling applications out of a single machine. Their key idea is to connect a host server with large memory pools [49, 58, 69] via fast datacenter network, which can be accessed by applications via OS kernel [14, 101, 113, 127] or software runtimes [53, 77, 106, 125, 126]. However, they do not provide cache coherence.

Distributed Programming Abstractions. Researchers have studied and proposed new programming languages and abstractions. Munin[21] built a type system that defines types for local and global pointers and tracks whether the pointer is shared via type checking. X10 [35, 57] and UPC [45] introduce function offloading interfaces for distributed computing and additional type annotations to reduce the runtime overhead. Ray [129] and Nu [107] are two recent systems proposing new abstractions for distributed programming. Unlike DRust, they require effort to port applications to avoid fine-grained memory sharing.

Hardware-Accelerated DSM. Specialized datacenter network technologies and emerging hardware designs stand for another trend to accelerate DSM. Clio [52], StRoM [117], and RMC [15] reduce remote memory access latency by offloading tasks into customized hardware. Concordia [128], Kona [29], and CXL 3.0 [39, 72, 73, 136] enable block-level or cache-line-level memory coherence with their hardware-implemented protocols. DRust can benefit from advances in hardware support and achieve better scalability.

5.8 Summary

This paper presents DRust, a practical DSM system based on the ownership model. It automatically turns a single-machine Rust program into its distributed version with a lightweight coherence protocol guided by language semantics. DRust significantly outperforms existing state-of-the-art DSM systems, demonstrating that a language-guided DSM can achieve strong memory consistency, transparency, and efficiency simultaneously.

CHAPTER 6

Conclusion and Future Directions

In this concluding chapter, we reflect on the key findings, discuss the broader implications of our work, and identify potential directions for future research.

6.1 Key Contributions

Resource disaggregation has emerged as a promising solution to enhance the efficiency of datacenters. However, existing disaggregation solutions often overlook the intrinsic semantics of the programs they support. This oversight results in missed optimization opportunities, adversely affecting performance in resource-disaggregated systems.

In fact, programs inherently contain rich semantic information which can be automatically extracted and utilized by the underlying systems for more informed decision-making. This dissertation introduces three innovative approaches to unearth and harness program semantics to facilitate the system design across multiple layers of the computing stack—from programming languages and compilers to runtime environments and operating systems. This work lays a foundational framework for the co-design and co-optimization of techniques across these varied layers, aimed at advancing future disaggregated data centers.

The three distinct approaches in this dissertation solve two primary challenges in existing disaggregated systems. The first two techniques, presented in Chapters 3 and 4, address the performance challenge in datacenter workloads running on memory-disaggregated clusters. These approaches leverage semantic information derived from the runtime's garbage collection

processes to build a more disaggregated-memory friendly runtime. In Chapter 3, we introduced Mako, a novel technique that offloads garbage collection tasks to memory servers. This approach not only reduces contention between application processing and garbage collection but also improves the efficiency of garbage collection operations by executing them closer to the data. Our experimental results demonstrated that Mako significantly improves performance in managed workloads running on disaggregated memory systems. In Chapter 4, we proposed MemLiner, a runtime technique that aligns garbage collection with the application's access patterns. By restructuring the tracing process in garbage collection to match application access patterns, MemLiner reduces interference and enhances data locality. This technique is proved effective in improving the performance of managed workloads in disaggregated memory systems. The third technique, detailed in Chapter 5, addresses the memory coherence challenge in compute-disaggregated systems. DRust leverages the ownership model to extract the SWMR semantics from applications. And with that semantics, it efficiently ensures coherence in compute-disaggregated systems.

6.2 Future Directions

The findings from this dissertation shows that by incorporating program semantics into system design, we can address longstanding performance and consistency challenges in resource-disaggregated systems. We conclude this dissertation with several open questions and potential future directions inspired by our findings.

Leveraging Ownership Semantics for Fault Tolerance in DSM systems. Fault tolerance is essential in distributed shared memory (DSM) systems to ensure reliability and availability across multiple nodes while preserving the illusion of shared memory. Traditional fault-tolerance approaches, such as full data replication or operation logging, can be costly due to the overhead of replicating or logging every change. A potentially more efficient solution involves leveraging ownership semantics to enable lightweight fault tolerance. The

ownership model captures object lifetimes and ownership transfers between threads, allowing for thread-level logging and replication without the constant overhead of replication or synchronization. For example, in the heap replication approach, global heap partitions are replicated at the same virtual address in backup servers, with threads operating on the primary heap partition and writing back to the backup only during ownership transfers. This strategy reduces network communication and the frequency of object replication.

Leveraging Semantics in LLM Workloads for GPU Memory Management. With the proliferation of large language models (LLMs), GPU resources have become the most critical asset in data centers. Efficiently managing GPU memory is increasingly important, especially since many large-model workloads, like transformer-based models used in inference serving, are memory-bound and require substantial GPU memory for Key-Value cache (KVCache). A future research avenue is to explore swapping some data to CPU memory, which is generally more cost-effective than GPU memory. This approach acts as a form of remote-memory based solution for GPU workloads, providing additional capacity for memory-bound processes. By leveraging CPU memory to augment GPU resources, data centers can achieve better GPU utilization and system efficiency. Additionally, customizing GPU memory management to fit specific workload characteristics can further enhance optimization. Given that LLMs typically rely on transformer-based architectures, this opens up opportunities to tailor memory management strategies to these unique usage patterns for improved efficiency and resource utilization.

Bibliography

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

[2] The Z garbage collector. https://wiki.openjdk.java.net/display/zgc/Main.

[3] Object table in Smalltalk. https://wiki.c2.com/?ObjectTable, 2004.

[4] The Rust programming language. http://www.rust-lang.org/, 2014.

[5] QuickCached. https://github.com/QuickServerLab/QuickCached, 2017.

[6] Memcached - a distributed memory object caching system. http://memcached.org, 2020.

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

[8] Apache cassandra: A open-source nosql database. https://cassandra.apache.org/_/index.html, 2021.

[9] Neo4j graph database. https://neo4j.com/, 2021.

[10] Lapack benchmark. https://www.netlib.org/lapack/lug/node71.html, 2023.

[11] Saleh E. Abdullahi and Graem A. Ringwood. Garbage collecting the internet: A survey of distributed garbage collection. *ACM Comput. Surv.*, 30(3):330–373, 1998.

[12] Marcos K. Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. Designing far memory data structures: Think outside the box. In *HotOS*, pages 120–126, 2019.

[13] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *ISCA*, pages 336–348, 2015.

[14] 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.

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

[16] Cristiana Amza, Alan L Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *Computer*, 29(2):18–28, 1996.

[17] Krste Asanovic. Firebox: A hardware building block for 2020 warehouse-scale computers. In *FAST*, 2014.

[18] Henry G. Baker. Lively linear lisp: "look ma, no garbage!". *SIGPLAN Not.*, 27 (8):89–98, aug 1992. ISSN 0362-1340. doi: 10.1145/142137.142162. URL https://doi.org/10.1145/142137.142162.

[19] Henry G. Baker, Jr. List processing in real time on a serial computer. *Commun. ACM*, 21(4):280–294, 1978.

[20] Thibaut Balabonski, François Pottier, and Jonathan Protzenko. The design and formalization of mezzo, a permission-based programming language. *ACM Trans. Program. Lang. Syst.*, 38(4), aug 2016. ISSN 0164-0925. doi: 10.1145/2837022. URL https://doi.org/10.1145/2837022.

[21] John K Bennett, John B Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the second ACM*

SIGPLAN symposium on Principles & practice of parallel programming, pages 168–176, 1990.

[22] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190, 2006.

[23] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[24] M. N. Bojnordi and E. Ipek. PARDIS: A programmable memory controller for the DDRx interfacing standards. In *ISCA*, pages 13–24, 2012.

[25] Mahdi Nazm Bojnordi and Engin Ipek. A programmable memory controller for the DDRx interfacing standards. *ACM Trans. Comput. Syst.*, 31(4):11:1–11:31, 2013.

[26] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. Theseus: an experiment in operating system structure and state management. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1–19. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/boos.

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

[28] 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*, pages 79–92, 2021.

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

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

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

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

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

[34] CCIX. Cache coherent interconnect for accelerators. https://www.ccixconsortium.com/, 2018.

[35] Satish Chandra, Vijay Saraswat, Vivek Sarkar, and Rastislav Bodik. Type inference for locality analysis of distributed data structures. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 11–22, 2008.

[36] Perry Cheng and Guy E Blelloch. A parallel, real-time garbage collector. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 125–136, 2001.

[37] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarría-Miranda. An evaluation of global address space languages: co-array fortran and unified parallel c. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 36–47, 2005.

[38] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[39] cxl. Compute express link 3.0. https://www.computeexpresslink.org/_files/ugd/0c1418_a8713008916044ae9604405d10a7773b.pdf, 2022.

[40] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, page 59–69, New York, NY, USA, 2001. Association for Computing Machinery. ISBN 1581134142. doi: 10.1145/378795.378811. URL https://doi.org/10.1145/378795.378811.

[41] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. In *ISMM*, pages 37–48, 2004.

[42] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. In *ISMM*, pages 37–48, 2004.

[43] dockerswarm. Managing a Cluster of Docker Daemons using Swarm Mode. https://docs.docker.com/engine/swarm/, 2023.

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

[45] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: distributed shared memory programming*. John Wiley & Sons, 2005.

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

[47] Christine H Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. Shenandoah: An open-source concurrent compacting garbage collector for openjdk. In *PPPJ*, pages 1–9, 2016.

[48] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362405. doi: 10.1145/3297858.3304013. URL https://doi.org/10.1145/3297858.3304013.

[49] GenZ. Genz consortium. http://genzconsortium.org/, 2019.

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

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

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

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

[54] David B Gustavson. The scalable coherent interface and related standards projects. *IEEE micro*, 12(1):10–22, 1992.

[55] h2oai. Database-like ops benchmark. https://github.com/h2oai/db-benchmark, 2023.

[56] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. Network support for resource disaggregation in next-generation datacenters. In *HotNets*, pages 10:1–10:7, 2013.

[57] Riyaz Haque and Jens Palsberg. Type inference for place-oblivious objects. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[58] Hewlett-Packard. The machine: A new kind of computer. https://www.hpl.hp.com/research/systems-research/themachine/.

[59] Richard L Hudson and J Eliot B Moss. Sapphire: Copying garbage collection without

stopping the world. *Concurrency and Computation: Practice and Experience*, 15(3-5): 223–261, 2003.

[60] Galen C. Hunt and James R. Larus. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, apr 2007. ISSN 0163-5980. doi: 10.1145/1243418.1243424. URL https://doi.org/10.1145/1243418.1243424.

[61] IBM. Daytrader. https://www.ibm.com/docs/en/linux-on-systems?topic=bad-daytrader, 2021.

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

[63] Intel. Intel high performance computing fabrics. https://www.intel.com/content/www/us/en/high-performance-computing-fabrics/, 2019.

[64] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *2002 USENIX Annual Technical Conference (USENIX ATC 02)*, Monterey, CA, June 2002. USENIX Association. URL https://www.usenix.org/conference/2002-usenix-annual-technical-conference/cyclone-safe-dialect-c.

[65] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011. ISBN 1420082795.

[66] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–34, 2017.

[67] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using rdma efficiently for key-value services. In *SIGCOMM*, pages 295–306, 2014.

[68] Stefanos Kaxiras, David Klaftenegger, Magnus Norgren, Alberto Ros, and Konstantinos Sagonas. Turning centralized coherence and distributed critical-section execution on their head: A new approach for scalable distributed shared memory. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 3–14, 2015.

[69] Kimberly Keeton. The Machine: An architecture for memory-centric computing. In *ROSS*, 2015.

[70] Haim Kermany and Erez Petrank. The compressor: Concurrent, incremental, and parallel compaction. In *PLDI*, pages 354–363, 2006.

[71] Bernard Lang and Francis Dupont. Incremental incrementally compacting garbage collection. In Richard L. Wexelblat, editor, *Proceedings of the Symposium on Interpreters and Interpretive Techniques, 1987, St. Paul, Minnesota, USA, June 24 - 26, 1987*, pages 253–263. ACM, 1987. doi: 10.1145/29650.29677. URL https://doi.org/10.1145/29650.29677.

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

[73] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 574–587, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399166. doi: 10.1145/3575693.3578835. URL https://doi.org/10.1145/3575693.3578835.

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

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

[76] 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*, pages 267–278, 2009.

[77] Haoran Ma, Shi Liu, Chenxi Wang, Yifan Qiao, Michael D. Bond, Stephen M. Blackburn, Miryung Kim, and Guoqing Harry Xu. Mako: A low-pause, high-throughput evacuating collector for memory-disaggregated datacenters. In *PLDI*, pages 92–107, 2022.

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

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

[80] Umesh Maheshwari and Barbara Liskov. Collecting distributed garbage cycles by back tracing. In *PODC*, pages 239–248, 1997.

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

[82] Mellanox. Connectx-6 single/dual-port adapter supporting 200gb/s with vpi. http://www.mellanox.com/page/products_dyn?product_family=265&mtag=connectx_6_vpi_card, 2019.

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

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

[85] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, David A. Wood, and Natalie Enright Jerger. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 2nd edition, 2020. ISBN 1681737094.

[86] Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Dependent types for imperative programs. *SIGPLAN Not.*, 43(9): 229–240, sep 2008. ISSN 0362-1340. doi: 10.1145/1411203.1411237. URL https://doi.org/10.1145/1411203.1411237.

[87] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. RedLeaf: Isolation and communication in a safe operating system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 21–39. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/narayanan-vikram.

[88] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. {Latency-Tolerant} software distributed shared memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 291–305, 2015.

[89] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *USENIX ATC*, pages 291–305, 2015.

[90] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian,

and Onur Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *OSDI*, pages 349–365, 2016.

[91] J Nieplocha, R Harrison, M Krishnan, B Palmer, and V Tipparaju. Combining shared and distributed memory models: Evolution and recent advancements of the global array toolkit. In *proceedings of POHLL'2002 workshop of ICS-2002, NYC*, 2002.

[92] Jaroslaw Nieplocha, Robert J Harrison, and Richard J Littlefield. Global arrays: A portable" shared-memory" programming model for distributed memory computers. In *Supercomputing'94: Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pages 340–349. IEEE, 1994.

[93] Jaroslaw Nieplocha, Robert J Harrison, and Richard J Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10:169–189, 1996.

[94] OpenCAPI. Open coherent accelerator processor interface. https://opencapi.org/, 2018.

[95] Oracle. Garbage first garbage collector tuning. https://www.oracle.com/technical-resources/articles/java/g1gc.html, 2020.

[96] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *NSDI*, pages 361–378, 2019.

[97] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The ramcloud storage system. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, August 2015. ISSN 0734-2071. doi: 10.1145/2806887. URL http://doi.acm.org/10.1145/2806887.

[98] Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. A study of concurrent real-time garbage collectors. *ACM SIGPLAN Notices*, 43(6):33–44, 2008.

[99] polars. Polars: Blazingly Fast DataFrame Library. https://pola-rs.github.io/polars/, 2023.

[100] Isabelle Puaut. A distributed garbage collector for active objects. In *OOPSLA*, pages 113–128, 1994.

[101] Yifan Qiao, Chenxi Wang, Zhenyuan Ruan, Adam Belay, Qingda Lu, Yiying Zhang, Miryung Kim, and Guoqing Harry Xu. Hermit:{Low-Latency},{High-Throughput}, and transparent remote memory via {Feedback-Directed} asynchrony. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 181–198, 2023.

[102] Yifan Qiao, Zhenyuan Ruan, Haoran Ma, Adam Belay, Miryung Kim, and Harry Xu. Harvesting idle memory for application-managed soft state with midas. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024.

[103] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. Understanding memory and thread safety practices and issues in real-world rust programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 763–779, 2020.

[104] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015. URL https://networkrepository.com.

[105] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-performance, application-integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 315–332. USENIX

Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/ruan.

[106] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. {AIFM}:{High-Performance},{Application-Integrated} far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 315–332, 2020.

[107] Zhenyuan Ruan, Seo Jin Park, Marcos K Aguilera, Adam Belay, and Malte Schwarzkopf. Nu: Achieving {Microsecond-Scale} resource fungibility with logical processes. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1409–1427, 2023.

[108] Stephen M. Rumble. Infiniband verbs performance. https://ramcloud.atlassian.net/wiki/display/RAM/Infiniband+Verbs+Performance, 2010.

[109] rust. Rust. https://www.rust-lang.org/, 2023.

[110] Narendran Sachindran, J Eliot B Moss, and Emery D Berger. Mc2: High-performance garbage collection for memory-constrained environments. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 81–98, 2004.

[111] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A user-programmable SSD. In *OSDI*, pages 67–80, 2014.

[112] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 323–337, 2017.

[113] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *OSDI*, pages 69–87, 2018.

[114] Aleksey Shipilev. TLAB allocation. https://shipilev.net/jvm/anatomy-quarks/4-tlab-allocation/, 2021.

[115] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. Shoal: A network architecture for disaggregated racks. In *NSDI*, pages 255–270, 2019.

[116] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. StRoM: Smart remote memory. In *EuroSys*, 2020.

[117] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. Strom: Smart remote memory. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450368827. doi: 10.1145/3342195.3387519. URL https://doi.org/10.1145/3342195.3387519.

[118] Robert Stets, Sandhya Dwarkadas, Nikolaos Hardavellas, Galen Hunt, Leonidas Kontothanassis, Srinivasan Parthasarathy, and Michael Scott. Cashmere-2l: Software coherent shared memory on a clustered remote-write network. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 170–183, 1997.

[119] Konstantin Taranov, Salvatore Di Girolamo, and Torsten Hoefler. Corm: Compactable remote memory over rdma. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1811–1824, 2021.

[120] Gil Tene, Balaji Iyengar, and Michael Wolf. C4: The continuously concurrent compacting collector. In *ISMM*, pages 79–88, 2011.

[121] Tokio Team. Build reliable network applications without compromising speed. https://tokio.rs/.

[122] John Toman, Stuart Pernsteiner, and Emina Torlak. Crust: A bounded verifier for rust (n). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 75–80, 2015. doi: 10.1109/ASE.2015.77.

[123] Maroun Tork, Lina Maudlej, and Mark Silberstein. Lynx: A SmartNIC-driven accelerator-centric architecture for network servers. In *ASPLOS*, pages 117–131, 2020.

[124] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–17, 2015.

[125] 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*, pages 261–280, 2020.

[126] Chenxi Wang, Haoran Ma, Shi Liu, Yifan Qiao, Jonathan Eyolfson, Christian Navasca, Shan Lu, and Guoqing Harry Xu. Memliner: Lining up tracing and application for a far-memory-friendly runtime. In *OSDI*, pages 35–53, 2022.

[127] Chenxi Wang, Yifan Qiao, Haoran Ma, Shi Liu, Yiying Zhang, Wenguang Chen, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Canvas: Isolated and adaptive swapping for multi-applications on remote memory. https://arxiv.org/abs/2203.09615, 2022.

[128] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. Concordia: Distributed shared memory with {In-Network} cache coherence. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 277–292, 2021.

[129] Stephanie Wang, Eric Liang, Edward Oakes, Ben Hindman, Frank Sifei Luan, Audrey Cheng, and Ion Stoica. Ownership: A distributed futures system for Fine-Grained tasks. In *18th USENIX Symposium on Networked Systems Design and Implementation*

(NSDI 21), pages 671–686. USENIX Association, April 2021. ISBN 978-1-939133-21-2. URL https://www.usenix.org/conference/nsdi21/presentation/cheng.

[130] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010. doi: 10.25080/Majora-92bf1922-00a.

[131] Mingyu Wu, Ziming Zhao, Yanfei Yang, Haoyu Li, Haibo Chen, Binyu Zang, Haibing Guan, Sanhong Li, Chuansheng Lu, and Tongbao Zhang. Platinum: A cpu-efficient concurrent garbage collector for tail-reduction of interactive services. In *USENIX ATC*, 2020.

[132] Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. Finding low-utility data structures. In *PLDI*, pages 174–186, 2010.

[133] Yahoo! Yahoo! cloud serving benchmark (YCSB). https://github.com/brianfrankcooper/YCSB, 2021.

[134] Taiichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3):181–198, 1990.

[135] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. HotCloud, page 10, Berkeley, CA, USA, 2010.

[136] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. Partial failure resilient memory management system for (cxl-based) distributed shared memory. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 658–674, 2023.