Mako: A Low-Pause, High-Throughput Evacuating Collector for Memory-Disaggregated Datacenters

Haoran Ma
University of California, Los Angeles
USA

Shi Liu
University of California, Los Angeles
USA

Chenxi Wang*
University of California, Los Angeles
USA

Yifan Qiao
University of California, Los Angeles
USA

Michael D. Bond
Ohio State University
USA

Stephen M. Blackburn
Australian National University
Australia

Miryung Kim
University of California, Los Angeles
USA

Guoqing Harry Xu*
University of California, Los Angeles
USA

Abstract

Resource disaggregation has gained much traction as an emerging datacenter architecture, as it improves resource utilization and simplifies hardware adoption. Under resource disaggregation, different types of resources (memory, CPUs, *etc.*) are disaggregated into dedicated servers connected by high-speed network fabrics. Memory disaggregation brings efficiency challenges to concurrent garbage collection (GC), which is widely used for latency-sensitive cloud applications, because GC and mutator threads simultaneously run and constantly compete for memory and swap resources.

Mako is a new concurrent and distributed GC designed for memory-disaggregated environments. Key to Mako's success is its ability to offload both tracing and evacuation onto memory servers and run these tasks concurrently when the CPU server executes mutator threads. A major challenge is how to let servers efficiently synchronize as they do not share memory. We tackle this challenge with a set of novel techniques centered around the *heap indirection table* (HIT), where entries provide one-hop indirection for heap pointers. Our evaluation shows that Mako achieves $\sim$**12*ms*** at the 90th-percentile pause time and outperforms Shenandoah by an average of **3**× in throughput.

CCS Concepts: • **Hardware** → **Communication hardware, interfaces and storage**; • **Software and its engineering** → **Garbage collection**; **Cloud computing**.

*Corresponding authors

PLDI '22, June 13–17, 2022, San Diego, CA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9265-5/22/06.
https://doi.org/10.1145/3519939.3523441

Keywords: Disaggregation, Memory Management, Garbage Collection, Cloud Computing

ACM Reference Format:
Haoran Ma, Shi Liu, Chenxi Wang, Yifan Qiao, Michael D. Bond, Stephen M. Blackburn, Miryung Kim, and Guoqing Harry Xu. 2022. Mako: A Low-Pause, High-Throughput Evacuating Collector for Memory-Disaggregated Datacenters. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3519939.3523441

1 Introduction

Resource disaggregation, as an emerging datacenter architecture, has recently attracted much attention in both academia [13, 32, 34, 59] and industry [3, 23, 27, 37, 45]. Resource-disaggregated datacenters are made up of servers dedicated to individual resource types (*e.g.*, CPU, memory, or accelerators), and connected by high-speed network fabrics such as RDMA over InfiniBand. Disaggregation is appealing due to three major advantages it provides: (1) *improved resource utilization*: decoupling resources and making them accessible to remote processes makes it easier for a job scheduler to achieve full resource utilization; (2) *improved failure isolation*: any server failure only reduces the amount of resource of a type, without affecting the availability of other types; and (3) *improved elasticity*: hardware-dedicated servers make it easy to adopt new hardware.

This paper focuses on an environment where memory is disaggregated [56, 66]—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 [10, 34, 59, 66] is through the paging/swap system

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

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 orders of magnitude [66].

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 [30] can slow down applications by 20×. Although concurrent GCs are necessary for latency-sensitive cloud applications [48], 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 paper presents Mako, a distributed and concurrent collector that achieves $\sim 12ms$ 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 $\sim$**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 [44]. 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 5–10*ms* 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 [66], 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.

2 Related Work

Disaggregation Systems. Due to rapid advances in networking hardware, resource disaggregation has quickly turned from a nebulous idea into a practical solution that is revolutionizing major datacenters [16, 20, 32, 35]. Given that resource underutilization is a major problem in modern datacenters, resource disaggregation attacks this problem head-on, turning resource scheduling from a difficult bin-packing problem into a much simpler allocation problem. Programs running on any CPU server can have their data located on any memory server. A disaggregated cluster also overcomes the critical 'memory capacity wall' problem [8, 11, 14, 41, 46, 47, 65] by allowing applications to use *unlimited* resources. The past few years have witnessed a proliferation of software systems that can take advantage of this architecture [7, 10, 25, 28, 29, 31, 34, 40, 43, 53, 56, 59, 61, 62].

Semeru [66] is a memory-disaggregated runtime that uses a unified address space for the distributed heap and offloads tracing to memory servers. However, Semeru was designed for throughput and, hence, object evacuation runs on the CPU server in a long stop-the-world (STW) phase. When evacuation runs, it must fetch objects from memory servers, move them onto the CPU server, and write them back to new locations on their memory servers. This process leads to exceedingly long GC pauses such as dozens of seconds, which are often unacceptable for latency-sensitive cloud services.

Concurrent GC. Many concurrent garbage collectors were designed to minimize GC pause times for cloud workloads that are more tolerant of reduced throughput (*i.e.*, longer end-to-end running time) than increased latency (*i.e.*, longer GC pauses). Oracle's Concurrent MarkSweep (CMS) [51] uses concurrent tracing to mark live objects (plus two short, STW phases) and concurrent sweeping to reclaim unmarked, dead objects. Many other concurrent moving collectors use concurrent tracing to mark live objects; their reclamation phase performs object evacuation by copying live objects from a from-space to a to-space. In doing so, they reap the benefits of reduced fragmentation, increased locality, and efficient bump pointer allocation, but at the cost of maintaining referential integrity while moving objects concurrently with mutator accesses.

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 [15], the Sapphire collector's blocking methods [39], CHICKEN and CLOVER's lock-free methods [54], and the Compressor collector's virtual memory protection mechanism [42]. Oracle's ZGC [1], RedHat's Shenandoah [30], and Azul's C4 [64] are widely used commercial concurrent moving garbage collectors, which use similar treatments to ensure safety in concurrent evacuation.

All of these concurrent moving collectors rely on synchronization techniques that apply only to cache-coherent shared-memory settings. These techniques do not work directly for memory disaggregation without huge performance penalties. For instance, by leveraging a cache coherence protocol, modern processors provide atomic instructions (*e.g.*, compare-and-swap) for threads to effectively synchronize over shared memory. However, these atomic instructions do not work when memory is distributed [36, 66] since network adapters typically do not provide cache coherence between servers—*e.g.*, if a memory server concurrently installs a forwarding pointer in an object, there does not exist a distributed "compare-and-swap" operation that can guarantee that the pointer is visible on the CPU server. As such, new synchronization techniques must be developed to support efficient atomicity between servers.

Other Moving and Non-moving GCs. There exists a large body of work on moving collectors that move live objects for memory efficiency. There are two major types of moving GCs: evacuating collectors [12, 15, 18, 21, 26, 38, 63, 65] that move live objects to a new space, reclaiming the old space *en masse*, and compacting collectors [24, 42, 57] that move live objects to one end of the same space, reclaiming the unused portion *en masse*. Moving collectors reduce fragmentation and enable the use of efficient allocators (*e.g.*, bump pointer), but incur either space (for evacuating collectors) or time overhead (for compacting collectors). Non-moving collectors typically perform sweep-to-freelist. They are efficient in time and space, but do not handle fragmentation well and must use a less efficient freelist-based allocator.

Ossia [52] adds a stop-the-world phase to the mark-sweep garbage collector that performs partial compaction for sparse regions in that phase. Immix [19] uses a combination of coarse-grained blocks and fine-grained lines for managing memory. It allows for contiguous allocation and resorts to opportunistic evacuation to defragment memory. Yak [50] combines a generational GC with region-based memory management for big data systems. NumaGiC [33] is a NUMA-aware GC that is designed to better exploit NUMA locality. Write-rationing GC [9] and Panthera [65] are moving collectors designed to efficiently migrate objects between DRAM and NVM. Unlike these moving collectors above, Mako has several unique characteristics optimized for memory disaggregation such as immediate reclamation, per-region evacuation, and fine-grained locking centered around the HIT.

3 Mako Overview

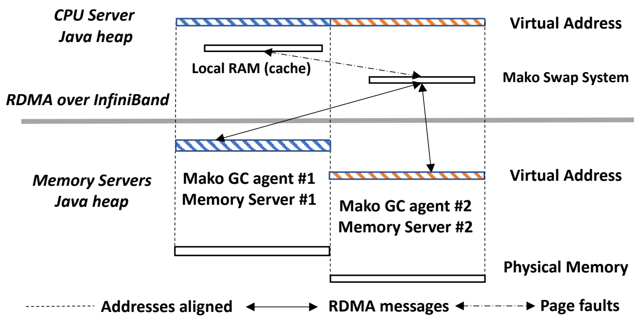


Figure 1. Mako's distributed Java heap.

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

3.1 Heap Structure

Figure 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.

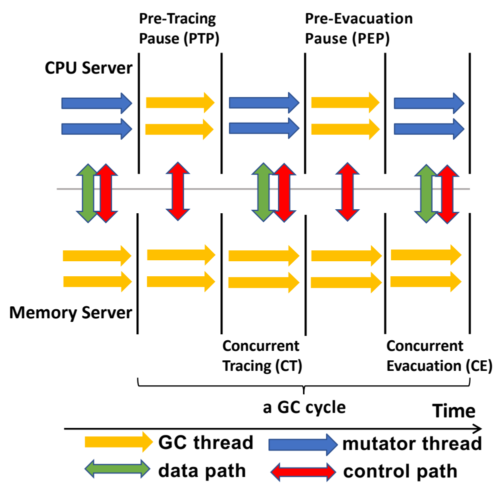
Servers are connected by RDMA over InfiniBand. Each memory server runs a Mako agent, which performs concurrent tracing and evacuation over local objects. This agent listens to the CPU server for commands as to what tasks to do and when to do them. Due to its simplicity, the Mako GC agent has a very short 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.

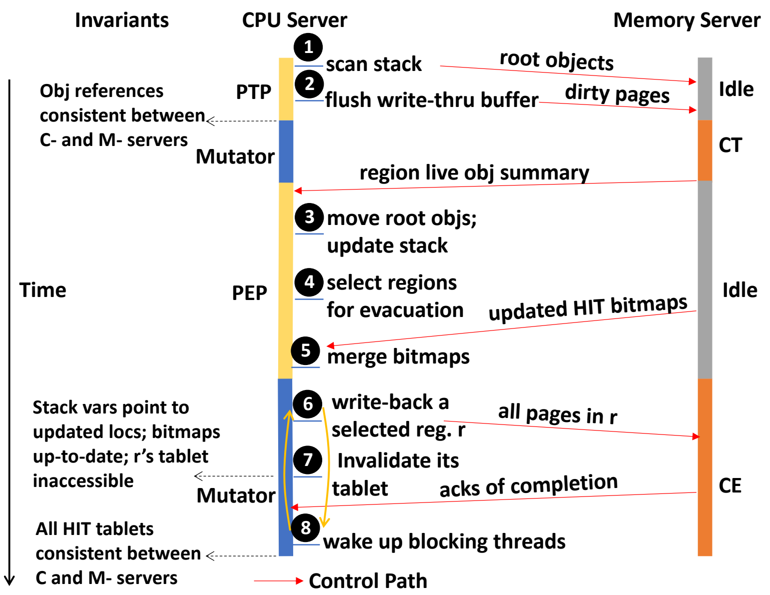
Mako uses a *region-based* heap, allowing us to perform concurrent object evacuation at the region granularity. When objects in a region are evacuated on a memory server, the CPU server can still access objects from other regions. Each region has a default size of 16MB, and the CPU server writes back a region if it is selected for evacuation on a memory server. Further details are discussed in §5. 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 Mako's Garbage Collector

Figure 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 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



(a) Phase overview



Invariants
CPU Server
Memory Server
1 scan stack
root objects
2 flush write-thru buffer
dirty pages
Idle
PTP
Obj references consistent between C- and M- servers
CT
Mutator
region live obj summary
3 move root objs; update stack
Time
PEP
4 select regions for evacuation
updated HIT bitmaps
Idle
5 merge bitmaps
Stack vars point to updated locs; bitmaps up-to-date; r's tablet inaccessible
6 write-back a selected reg. r
all pages in r
7 Invalidate its tablet
acks of completion
Mutator
CE
All HIT tablets consistent between C and M- servers
8 wake up blocking threads
Control Path

(b) GC protocol, control-path communication, and invariants

Figure 2. An overview of Mako's concurrent GC.

Evacuation (CE) are concurrent phases run by each memory server, as the CPU server runs the mutator. Figure 2(b) illustrates the main activities in each phase, as elaborated below:

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 [68], 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. 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).

[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.

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 §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 [2]. 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 §4.

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 §5.3.

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 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 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

4 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: (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.

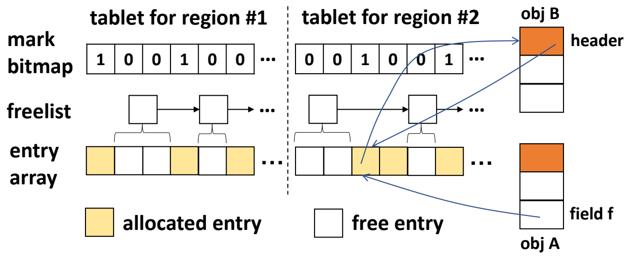


Figure 3. The structure of the heap indirection table (HIT). As an example, the field f of object A references object B.

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 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 §5.1). 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 [60]. 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 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.

5 GC Design

5.1 Barriers

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).

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

```
1  Function LoadBarrier(a = b.f)
2      HIT entry $e \leftarrow b.f$ ;
3      if CE_RUNNING then
4          Region $r \leftarrow$ 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                  $t \leftarrow$ Move($b.f$, $r'$);
9                  Atomic {
10                 /* only one thread can update $*e$ */
11                 if Region($*e$)$\neq r'$ then
12                     $*e \leftarrow t$ ;
13                 }
14             else
15                 /* $r$ is being evacuated on a mem server */
16                 while $\neg$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     $a \leftarrow *e$ ;
20 Function StoreBarrier(b.f = a)
21     /* obtain the entry address from $a$'s header */
22     HIT entry $e \leftarrow$Entry($a$);
23     $b.f \leftarrow e$ ;
```

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 (§5.3) 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). IsValid($r.tablet$) returns false if r's tablet is invalidated by the GC thread (Line 14 in Algorithm 2) 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 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.

5.2 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 [68] 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 [68].

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 [6, 49, 55]. 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.

5.3 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 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

Algorithm 2: PEP and CE.

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

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 13). 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 14); 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 16) 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 18) and all pages in the `to-space` r' (Line 19). 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 33). 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 34). Upon receiving a message (Line 23), the GC thread on the CPU server unregisters r (Line 27) 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 26) so that mutators threads blocking on r can continue (Line 16). We modify the object allocator to *not* allocate into regions

Table 2. Systems and applications used to evaluate Mako.

DaCapo [17]	**Size**	
Tradesoap (**DTS**)	DaCapo/huge	
Tradebeans (**DTB**)	DaCapo/huge	
H2 (**DH2**)	DaCapo/huge	
Apache Cassandra [4]	**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 [69]	**Dataset & Size**	
PageRank (**SPR**)	Wikipedia Polish [5] (1 GB)	
Transitive Closure (**STC**)	Generated Graph (1.5M edges, 384K vertices)	

in the evacuation set. Hence, allocation will never block on concurrent evacuation.

6 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 [17], and several applications on Cassandra [4] (a NoSQL columnar database) and Spark [69] (a de-facto big data analytics engine), as shown in Table 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 [67] 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 [30], a modern concurrent collector in OpenJDK, and Semeru [66], 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 [1], 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 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 [10, 56, 59, 66].

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.

6.1 Throughput (End-to-End Performance)

Figure 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.

Semeru [66] 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 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 §6.2, 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

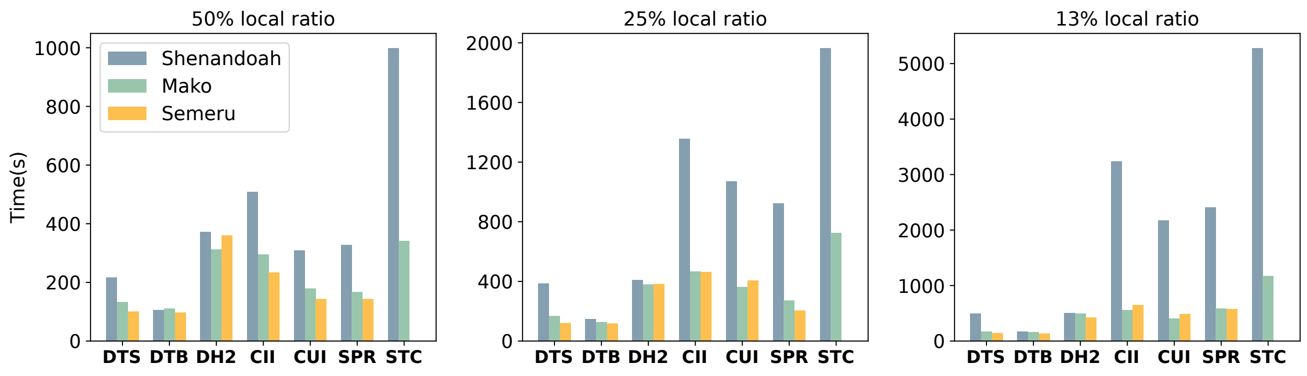


Figure 4. End-to-end time under Shenandoah GC [30], Semeru [66] and Mako for 50%, 25% and 13% local memory ratios. Semeru crashed when running **STC** so its bars are not shown.

Table 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

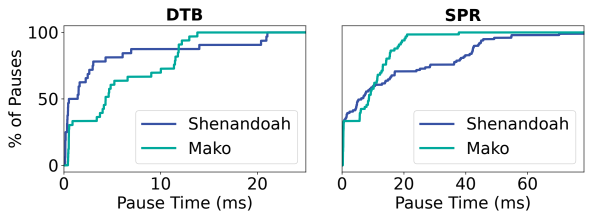


Figure 5. Pause time CDF for **DTB** and **SPR**.

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.

6.2 GC Latency

This section compares Mako's pause time with Shenandoah and Semeru. Table 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 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 **11*ms*** and **18*ms*** vs. Shenandoah's **14*ms*** and **42*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.

To better understand the distribution of collection pauses, we additionally report the *bounded minimum mutator utilization* (BMU) for **DTB** and **SPR** in Figure 6. The minimum mutator utilization (MMU) was defined by Cheng and Blelloch [22] as the minimum fraction of the mutator's execution time within any window of a specified size. Sachindran *et al.* [58] 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 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

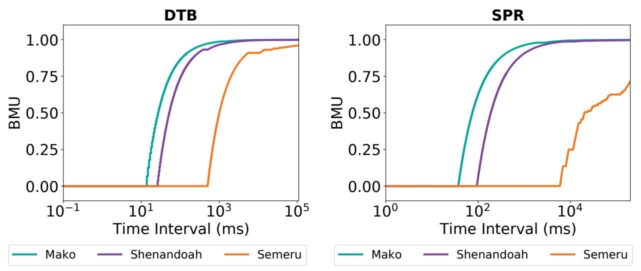


Figure 6. Bounded minimum mutator utilization.

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.

6.3 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 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 §6.1, these improvements are much larger than the barrier-incurred overheads.

Table 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 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 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 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 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.

6.4 Collection Effectiveness

The goal of this experiment is to compare the memory reclamation effectiveness of Shenandoah, Semeru, and Mako. Figure 7 shows the pre-GC and after-GC memory footprints under the 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 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.

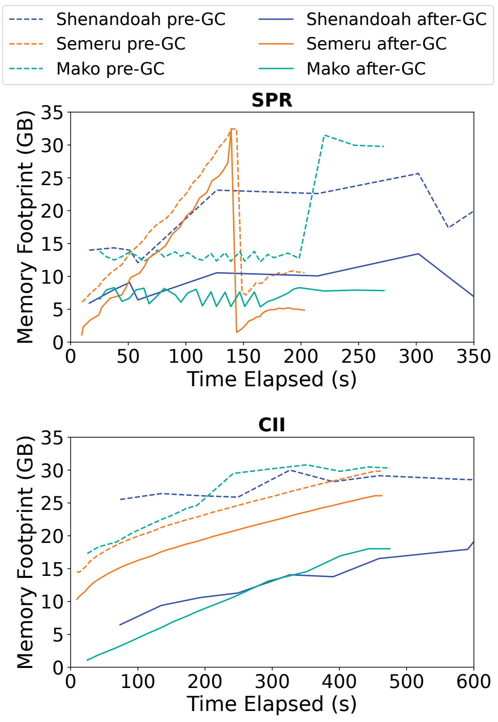


Figure 7. GC effectiveness under 25% cache ratio.

6.5 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 15.32*ms* to 8.13*ms*). 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 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.

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.

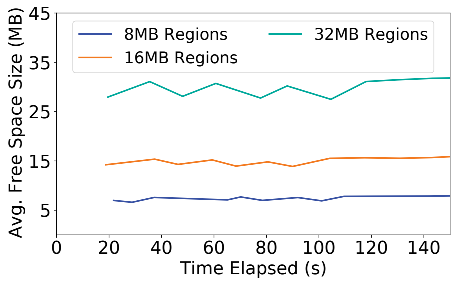


Figure 8. The average size of the intra-region contiguous free space for different region sizes.

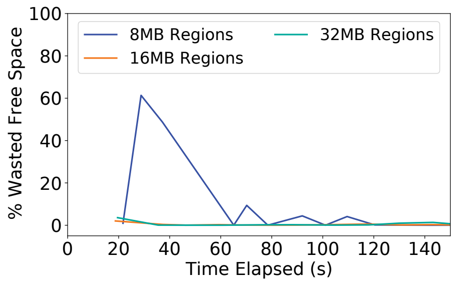


Figure 9. Ratio of wasted free space over total heap usage for different region sizes.

To quantify this waste, we report the ratio between the sizes of the wasted space and the used heap in Figure 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 10.1*ms* GC pause time and 272.71s throughput.

7 Conclusion

This paper presents Mako, the first concurrent evacuating collector that provides low pause times for the emerging datacenter architecture with memory disaggregation. Mako 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.

Acknowledgments

We thank the PLDI reviewers for their comments. We are grateful to our shepherd Sean Treichler for his feedback. This work is supported by NSF grants NSF XPS-1629126, CCF-1764077, CCF-1723773, CNS-1703598, CNS-1907352, CHS-1956322, CNS-2007737, CNS-2006437, CSR-2106117, CNS-2106838, CCF-2106404, and CNS-2128653, ONR grant N00014-18-1-2037, Australian ARC grant DP190103367, as well as grants from Intel, Samsung, and Cisco.

References

[1] [n.d.]. The Z Garbage Collector. https://wiki.openjdk.java.net/display/zgc/Main.

[2] 2004. Object Table in Smalltalk. https://wiki.c2.com/?ObjectTable.

[3] 2010. SeaMicro Technology Overview. https://data.tiger-optics.ru//download/seamicro/SM_TO02_v1.4.pdf.

[4] 2021. Apache Cassandra: A open-source NoSQL Database. https://cassandra.apache.org/_/index.html.

[5] 2021. KONECT Network Datasets. http://konect.cc/networks/.

[6] Saleh E. Abdullahi and Graem A. Ringwood. 1998. Garbage Collecting the Internet: A Survey of Distributed Garbage Collection. *ACM Comput. Surv.* 30, 3 (1998), 330–373.

[7] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2018. Remote Regions: A Simple Abstraction for Remote Memory. In *USENIX ATC*. 775–787.

[8] Marcos K. Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. 2019. Designing Far Memory Data Structures: Think Outside the Box. In *HotOS*. 120–126.

[9] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. 2018. Write-Rationing Garbage Collection for Hybrid Memories. In *PLDI*. 62–77.

[10] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can Far Memory Improve Job Throughput?. In *EuroSys*. Article 14.

[11] Sebastian Angel, Mihir Nanavati, and Siddhartha Sen. 2020. Disaggregation and the Application. In *HotCloud*.

[12] Andrew Appel. 1989. Simple generational garbage collection and fast allocation. *Software Practice & Experience* (1989), 171–183.

[13] Krste Asanovic. 2014. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers. In *FAST*.

[14] Krste Asanović, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. 2006. *The Landscape of Parallel Computing Research: A View from Berkeley*. Technical Report UCB/EECS-2006-183. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[15] Henry G. Baker. 1978. List Processing in Real Time on a Serial Computer. *Commun. ACM* 21, 4 (April 1978), 280–294.

[16] Luiz Andre Barroso. 2011. Warehouse-Scale Computing: Entering the Teenage Decade. In *ISCA*.

[17] 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. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA*. 169–190.

[18] Stephen M Blackburn, Richard Jones, Kathryn S. McKinley, and J Eliot B Moss. 2002. Beltway: Getting around Garbage Collection Gridlock. In *PLDI*. 153–164.

[19] Stephen M. Blackburn and Kathryn S. McKinley. 2008. Immix: A Mark-Region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance. In *PLDI*. 22–32.

[20] Amanda Carbonari and Ivan Beschasnikh. 2017. Tolerating Faults in Disaggregated Datacenters. In *HotNets-XVI*. 164–170.

[21] C. J. Cheney. 1970. A Nonrecursive List Compacting Algorithm. *Commun. ACM* 13, 11 (Nov. 1970), 677–678.

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

[23] I-Hsin Chung, Bulent Abali, and Paul Crumley. 2018. Towards a Composable Computer System. In *HPC Asia*. 137–147.

[24] Jacques Cohen and Alexandru Nicolau. 1983. Comparison of Compacting Algorithms for Garbage Collection. *ACM Trans. Program. Lang. Syst.* 5, 4 (1983), 532–553.

[25] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. 1994. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *OSDI*.

[26] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-First Garbage Collection. In *ISMM*. 37–48.

[27] Facebook and Intel. 2013. Facebook and Intel Collaborate on Future Data Center Rack Technologies. http://goo.gl/6h2Ut.

[28] M. J. Feeley, W. E. Morgan, E. P. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. 1995. Implementing Global Memory Management in a Workstation Cluster. In *SOSP*. 201–212.

[29] E. Felten and J. Zahorjan. 1991. Issues in the implementation of a remote memory paging system. In *University of Washington CSE TR CSE TR*.

[30] Christine H. Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. 2016. Shenandoah: An Open-source Concurrent Compacting Garbage Collector for OpenJDK. In *PPPJ*. 13:1–13:9.

[31] Michail D. Flouris and Evangelos P. Markatos. 1999. The Network RamDisk: Using remote memory on heterogeneous NOWs. *Cluster Computing* 2, 4 (01 Dec 1999).

[32] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network Requirements for Resource Disaggregation. In *OSDI*. 249–264.

[33] Lokesh Gidra, Gaël Thomas, Julien Sopena, Marc Shapiro, and Nhan Nguyen. 2015. NumaGiC: A Garbage Collector for Big Data on Big NUMA Machines. In *ASPLOS*. 661–673.

[34] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with INFINISWAP. In *NSDI*. 649–667.

[35] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. 2013. Network Support for Resource Disaggregation in Next-generation Datacenters. In *HotNets*. 10:1–10:7.

[36] J. Hennessy, M. Heinrich, and A. Gupta. 1999. Cache-coherent distributed shared memory: perspectives on its development and future challenges. *Proc. IEEE* 87, 3 (1999), 418–429. https://doi.org/10.1109/5.747863

[37] Hewlett-Packard. [n.d.]. The Machine: A New Kind of Computer. https://www.hpl.hp.com/research/systems-research/themachine/.

[38] Richard L. Hudson and J. Eliot B. Moss. 1992. Incremental Collection of Mature Objects. In *ISMM*. 388–403.

[39] Richard L Hudson and J Eliot B Moss. 2003. Sapphire: Copying garbage collection without stopping the world. *Concurrency and Computation: Practice and Experience* 15, 3-5 (2003), 223–261.

[40] L. Iftode, K. Li, and K. Petersen. 1993. Memory servers for multicomputers. In *Digest of Papers. Compcon Spring*. 538–547. https://doi.org/10.1109/CMPCON.1993.289731

[41] Kimberly Keeton. 2015. The Machine: An Architecture for Memory-Centric Computing. In *ROSS*.

[42] Haim Kermany and Erez Petrank. 2006. The Compressor: Concurrent, Incremental, and Parallel Compaction. In *PLDI*. 354–363.

[43] S. Koussih, A. Acharya, and S. Setia. 1999. Dodo: a user-level system for exploiting idle memory in workstation clusters. In *HPDC*. 301–308.

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

[45] Sergey Legtchenko, Hugh Williams, Kaveh Razavi, Austin Donnelly, Richard Black, Andrew Douglas, Nathanaël Cheriere, Daniel Fryer, Kai

Mast, Angela Demke Brown, Ana Klimovic, Andy Slowey, and Antony Rowstron. 2017. Understanding Rack-scale Disaggregated Storage. In *HotStorage*.

[46] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. 2009. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *ISCA*. 267–278.

[47] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. 2012. System-level implications of disaggregated memory. In *HPCA*. 1–12.

[48] Martin Maas, Krste Asanović, Tim Harris, and John Kubiatowicz. 2016. Taurus: A Holistic Language Runtime System for Coordinating Distributed Managed-Language Applications. In *ASPLOS*. 457–471.

[49] Umesh Maheshwari and Barbara Liskov. 1997. Collecting Distributed Garbage Cycles by Back Tracing. In *PODC*. 239–248.

[50] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. 2016. Yak: A High-Performance Big-Data-Friendly Garbage Collector. In *OSDI*. 349–365.

[51] Oracle. 2001. Concurrent MarkSweep GC. https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/cms.html.

[52] Yoav Ossia, Ori Ben-Yitzhak, and Marc Segal. 2004. Mostly concurrent compaction for mark-sweep GC. In *Proceedings of the 4th international symposium on Memory management*. 25–36.

[53] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *NSDI*. 361–378.

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

[55] Isabelle Puaut. 1994. A Distributed Garbage Collector for Active Objects. In *OOPSLA*. 113–128.

[56] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: High-Performance, Application-Integrated Far Memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 315–332. https://www.usenix.org/conference/osdi20/presentation/ruan

[57] Narendran Sachindran, J. Eliot B. Moss, and Emery D. Berger. 2004. MC2: High-Performance Garbage Collection for Memory-Constrained Environments. In *OOPSLA*. 81–98.

[58] Narendran Sachindran, J Eliot B Moss, and Emery D Berger. 2004. 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*. 81–98.

[59] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *OSDI*. 69–87.

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

[61] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. 2019. Shoal: A Network Architecture for Disaggregated Racks. In *NSDI*. 255–270.

[62] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. 2020. StRoM: Smart Remote Memory. In *EuroSys*. Article 29.

[63] Darko Stefanović, Kathryn S. McKinley, and J. Eliot B. Moss. 1999. Age-Based Garbage Collection. In *OOPSLA*. 370–381.

[64] Gil Tene, Balaji Iyengar, and Michael Wolf. 2011. C4: The Continuously Concurrent Compacting Collector. In *ISMM*. 79–88.

[65] Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. 2019. Panthera: Holistic Memory Management for Big Data Processing over Hybrid Memories. In *PLDI*. 347–362.

[66] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2020. Semeru: A Memory-Disaggregated Managed Runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 261–280. https://www.usenix.org/conference/osdi20/presentation/wang

[67] Yahoo! 2021. Yahoo! Cloud Serving Benchmark (YCSB). https://github.com/brianfrankcooper/YCSB.

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

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