Semeru: A Memory-Disaggregated Managed Runtime

Chenxi Wang[†] Haoran Ma[†] Shi Liu[†] Yuanqi Li[†] Zhenyuan Ruan[‡] Khanh Nguyen[§]
Michael D. Bond[*] Ravi Netravali[†] Miryung Kim[†] Guoqing Harry Xu[†]
UCLA[†] *MIT*[‡] *Texas A&M University*[§] *Ohio State University*[*]

Abstract

Resource-disaggregated architectures have risen in popularity for large datacenters. However, prior disaggregation systems are designed for native applications; in addition, all of them require applications to possess excellent locality to be efficiently executed. In contrast, programs written in managed languages are subject to periodic garbage collection (GC), which is a typical graph workload with poor locality. Although most datacenter applications are written in managed languages, current systems are far from delivering acceptable performance for these applications.

This paper presents *Semeru*, a distributed JVM that can dramatically improve the performance of managed cloud applications in a memory-disaggregated environment. Its design possesses three major innovations: (1) a universal Java heap, which provides a unified abstraction of virtual memory across CPU and memory servers and allows any legacy program to run *without modifications*; (2) a distributed GC, which offloads *object tracing* to memory servers so that tracing is performed *closer to data*; and (3) a swap system in the OS kernel that works with the runtime to swap page data efficiently. An evaluation of *Semeru* on a set of widely-deployed systems shows very promising results.

1 Introduction

The idea of *resource disaggregation* has recently attracted a great deal of attention in both academia [16, 45, 49, 87] and industry [3, 33, 39, 52, 65]. Unlike conventional datacenters that are built with *monolithic* servers, each of which tightly integrates a small amount of each type of resource (*e.g.*, CPU, memory, and storage), resource-disaggregated datacenters contain servers dedicated to individual resource types. Disaggregation is particularly appealing due to three major advantages it provides: (1) *improved resource utilization*: decoupling resources and making them accessible to remote processes make it much easier for a job scheduler to achieve full resource utilization; (2) *improved 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) *improved elasticity*: hardware-dedicated servers make it easy to adopt and add new hardware.

State of the Art. Architecture [10, 22, 23, 58] and networking [7, 30, 46, 55, 72, 83, 86, 88] technologies have matured to a point at which data transfer between servers is fast enough for them to execute programs collectively. LegoOS [87] provides a new OS model called *splitkernel*, which disseminates traditional OS components into loosely coupled monitors, each of which runs on a resource server. InfiniSwap [49] is a paging system that leverages RDMA to expose memory to applications running on remote machines. FaRM [37] is a distributed memory system that uses RDMA for both fast messaging and data access. There also exists a body of work [12, 28, 38, 60, 61, 64, 65, 73, 77, 94, 96, 97, 105] on storage disaggregation.

1.1 Problems

Although RDMA provides efficient data access among remote access techniques, fetching data from remote memory on a memory-disaggregated architecture, is time consuming, incurring microsecond-level latency that cannot be handled well by current system techniques [20]. While various optimizations [37, 38, 49, 84, 87, 105] have been proposed to reduce or hide fetching latency, such techniques focus on the low-level system stack and do *not* consider *run-time semantics* of a program, such as locality.

Improving performance for applications that exhibit *good locality* is straightforward: the CPU server runs the program, while data are located on memory servers; the CPU server has only a small amount of memory used as a *local cache*[1] that stores recently fetched pages. A cache miss triggers a page fault on the CPU server, making it fetch data from the memory server that hosts the requested page. Good locality reduces cache misses, leading to improved application performance. As a result, a program itself needs to possess *excellent spatial and/or temporal locality* to be executed efficiently under current memory-disaggregation systems [7, 8, 49, 87].

This high requirement of locality creates two practical challenges for cloud applications. First, typical cloud applications are written in managed languages that execute atop a managed runtime. The runtime performs automated memory management using *garbage collection (GC)*, which frequently traces the heap and reclaims unreachable objects. GC is a typical graph workload that performs reachability analysis over a huge graph of objects connected by references. Graph traversal often suffers from poor locality, so GC running on the CPU server potentially triggers a page fault as it follows each reference. As shown in §2, memory disaggregation can increase the duration of GC pauses by $>10\times$, significantly degrading application performance.

[1] In this paper, "cache" refers to local memory on the CPU server.

Second, to make matters worse, unlike native programs whose data structures are primarily array-based, managed programs make heavy use of object-oriented data structures [74, 100, 101], such as maps and lists connected via pointers without good locality. To illustrate, consider a Spark RDD — it is essentially a large list that references a huge number of element objects, which can be distributed across memory servers. Even a sequential scan of the list needs to access arbitrarily located elements, incurring high performance penalties due to frequent remote fetches.

In essence, managed programs such as Spark, which are typical cloud workloads that resource disaggregation aims to benefit, have not yet received much support from existing resource-disaggregated systems.

1.2 Our Contributions

Goal and Insight. The goal of this project is to design a memory-disaggregation-friendly managed runtime that can provide superior efficiency to all managed cloud applications running in a memory-disaggregated datacenter. Our major drive is an observation that shifting our focus from low-level, semantics-agnostic optimizations (as done in prior work) to the *redesign of the runtime* that improves data placement, layout, and usage, can unlock massive opportunities.

To achieve this goal, our insights are as follows. To exploit *locality for GC*, most GC tasks can be *offloaded* to memory servers where data is located. As GC tasks are mostly memory intensive, this offloading fits well into a memory server's resource profile: weak compute and abundant memory. Memory servers can perform some offloaded GC tasks — such as tracing objects — *concurrently* with application execution. Similarly, other GC tasks — such as evacuating objects and reclaiming memory — can be offloaded to memory servers, albeit while application execution is paused. Furthermore, evacuation can improve *application locality* by moving objects likely to be accessed together to contiguous memory.

Semeru. Following these insights, we develop *Semeru*,[2] a distributed Java Virtual Machine (JVM) that supports efficient execution of *unmodified* managed applications. As with prior work [49, 87], this paper assumes a setting where processes on each CPU server can use memory from multiple memory servers, but no single process spans multiple CPU servers. *Semeru*'s design sees three major challenges:

The first challenge is what memory abstraction to provide. A reachability analysis over objects on a memory server requires the server to run a user-space process (such as a JVM) that has its own address space. As such, the same object may have different virtual addresses between the CPU server (that runs the main process) and its hosting memory server (that runs the tracing process). Address translation for each object can incur large overheads.

To overcome this challenge, *Semeru* provides a memory abstraction called the *universal Java heap (UJH)* (§3.1). The execution of the program has a main compute process running on the CPU server as well as a set of "assistant" processes, each running on a memory server. The main and assistant processes are all JVM instances, and servers are connected with RDMA over InfiniBand. The main process executes the program while each assistant process only runs offloaded memory management tasks. The heap of the main process sees a contiguous virtual address space partitioned across the participating memory servers, each of which sees and manages a disjoint range of the address space. *Semeru* enables an object to have the same virtual address on both the CPU server and its hosting memory server, making it easy to separate an application execution from the GC tasks.

The second challenge is what to offload. An ideal approach is to run the entire GC on memory servers while the CPU server executes the program, so that memory management tasks are performed (1) near data, providing locality benefits, and (2) concurrently without interrupting the main execution. However, this approach is problematic because some GC operations — notably evacuating (moving) and compacting objects into a new region — must coordinate extensively with application threads to preserve correctness. As a result, many GC algorithms — including the high-performance GC that our work extends — trace live objects concurrently with application execution, but move objects only while application execution is paused (*i.e.*, stop-the-world collection).

We develop a distributed GC (§4) that selectively offloads tasks and carefully coordinates them to maximize GC performance. Our idea is to offload *tracing* to memory servers concurrently with application execution. Tracing computes a transitive closure of live objects from a set of roots. It does nothing but pointer chasing, which would be a major bottleneck if performed at the CPU server. To avoid this bottleneck, *Semeru* lets each memory server trace its own objects, as opposed to bringing them into the CPU server for tracing.

Tracing is a memory-intensive task that does not need much compute [27] but benefits greatly from being close to data. To leverage memory servers' weak compute, memory servers trace their local objects *continuously* while the CPU server executes the main threads. Tracing also fits well into various hardware accelerators [69, 85], which future memory servers may employ. The CPU server periodically stops the world for memory servers to *evacuate* live objects (*i.e.*, copy them from old to new memory regions) to reclaim memory. Object evacuation provides a unique opportunity for *Semeru* to relocate objects that may potentially be accessed together into a *contiguous* space, improving spatial locality.

The third challenge is how to efficiently swap data. Existing swap systems such as InfiniSwap [49] and FastSwap [11] cannot coordinate with the language runtime and have bugs when running distributed frameworks such as Spark (§2). Mellanox provides an NVMe-over-fabric (NVMe-oF) [1] driver that allows the CPU server to efficiently access remote storage using RDMA. A strawman approach here is to mount

[2] *Semeru* is the highest mountain on the island of East Java.

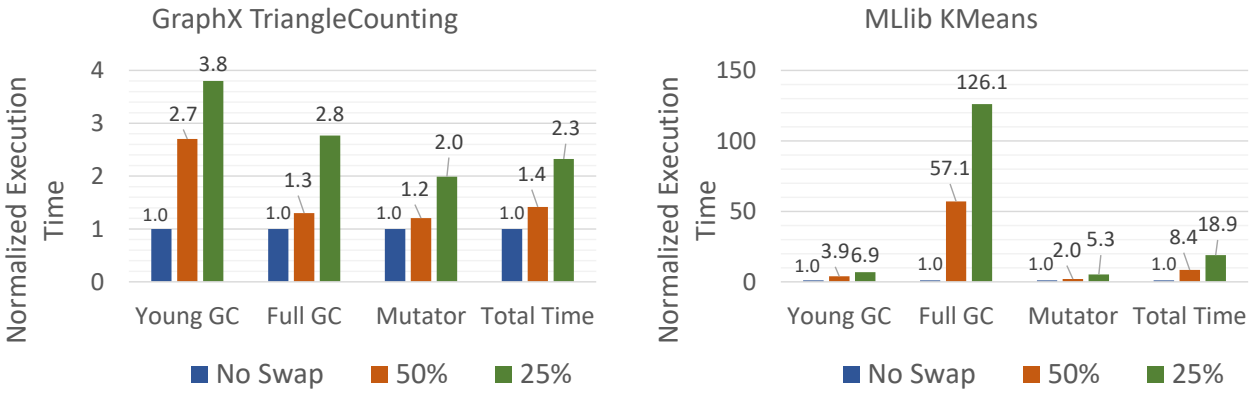


Figure 1: Slowdowns of two representative Spark applications under disaggregated memory; NVMe-oF was used for data swapping. Spark was executed over OpenJDK 12 with its default (Garbage First) GC. The four groups for each program report the slowdowns of the nursery (young) GC, full-heap GC, mutator, and end-to-end execution. Each group contains three bars, reporting the execution times under three cache configurations: 100%, 50%, and 25%. Each configuration represents a percentage of the application's working set that can fit into the CPU server's local DRAM. Execution times of the 50% and 25% configurations are normalized to that of 100%.

remote memory as RAMDisks and use NVMe-oF to swap data. However, this approach does not work in our setting where remote memory is subject to memory-server tracing and compaction, precluding it from being used as RAMDisks. To this end, we modify the NVMe-oF implementation (§5) to provide support for remote memory management. InfiniBand gather/scatter is used to efficiently transfer pages. We also develop new system calls that enable effective communications between the runtime and the swap system.

Results. We have evaluated *Semeru* using two widely-deployed systems – Spark and Flink – each with a representative set of programs. Our results demonstrate that *Semeru* improves the end-to-end performance of these systems by an average of **2.1**× and **3.7**× when the cache size is 50% and 25% of the heap size, application performance by an average of **1.9**× and **3.3**×, and GC performance by **4.2**× and **5.6**×, respectively, compared to running these systems directly on NVM-oF where remote accesses incur significant latency overheads. These promising results suggest that *Semeru* reduces the gap between memory disaggregation and managed cloud applications, taking a significant step toward efficiently running such applications on disaggregated datacenters.

Semeru is publicly available at `https://github.com/uclasystem/Semeru`.

2 Motivation

We conducted experiments to understand the latency penalties that managed programs incur on existing disaggregation systems. We first tried to use existing disaggregation systems including LegoOS [87], InfiniSwap [49], and FastSwap [11]. However, LegoOS does not yet support socket system calls and cannot run socket-based distributed systems such as Spark. Under InfiniSwap and FastSwap, the JVM was frequently stuck — certain remote fetches never returned.

Background of G1 GC. To collect preliminary data, we set up a small cluster with one CPU and two memory servers, using Mellanox's NVMe-over-fabric (NVMe-oF) [1] protocol for data swapping, mounting remote memory as a RAMDisk. On this cluster, we ran two representative Spark applications: Triangle Counting (TC) from GraphX and KMeans from MLib with the Twitter graph [63] as the input. We used OpenJDK 12 with its high-performance *Garbage First (G1)* GC, which is the default GC recommended for large-scale processing tasks, with a 32GB heap. G1 is a *region-based, generational* GC that most frequently traces the young generation (*i.e.*, nursery GC) and occasionally traces both young and old generations (*i.e.*, full-heap GC). This is based on the *generational hypothesis* that most objects die young and hence the young generation contains a larger fraction of garbage than the old generation [93].

Under G1, the memory for both the young and old generations is divided into *regions*, each being a contiguous range of address space. Objects are allocated into regions. Each nursery GC traces a small number of selected regions in the young generation. After tracing, live objects in these regions are evacuated (*i.e.*, moved) into new regions. Objects that have survived a number of nursery GCs will be *promoted* to the old generation and subject to less frequent tracing. Each full-heap GC traces the entire heap, and then evacuates and compacts a subset of regions.

Performance. The performance of these applications is reported in Figure 1. In particular, we measured time spent on nursery and full-heap collections, as well as end-to-end execution time. Three cache configurations (shown in three bars of each group) were considered, each representing a particular percentage of the application's working set that can fit into the CPU server's local DRAM.

Despite the many block-layer optimizations in the NVMe-oF swap system, performance penalties from remote fetching are still large. Under the 25% cache configuration, the average slowdown for these applications is **10.6**×. Note that for a typical Big Data application with a large working set (*e.g.*, 80–100GB), 25% of the working set means that the CPU server

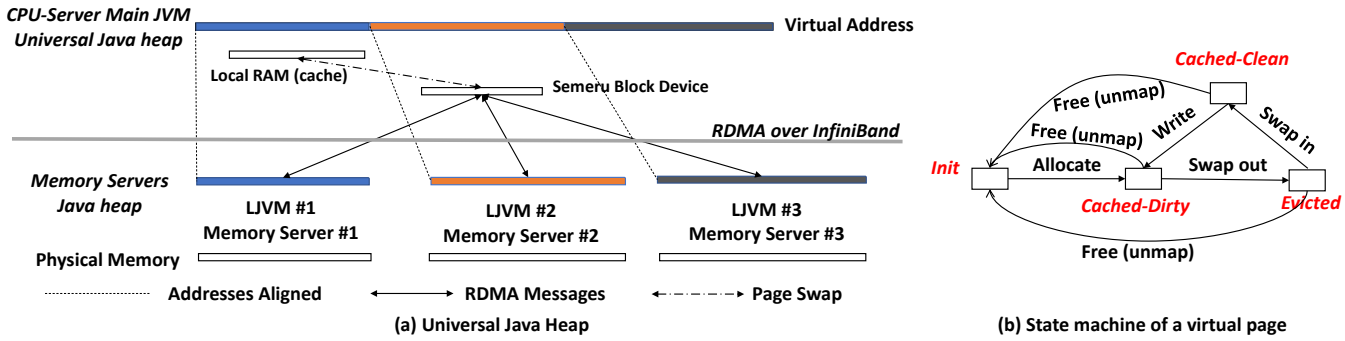


Figure 2: *Semeru*'s heap and virtual page management.

needs at least 20–25GB DRAM for *a single application* to have a ∼10× slowdown. Considering a realistic setting where the CPU server runs multiple applications, there is a much higher DRAM requirement for the CPU server, posing a practical challenge for disaggregation.

Takeaway. Disaggregated memory incurs a higher slowdown for the GC than the main application threads (*i.e.*, *mutator* threads in GC literature terminology) — this is easy to understand because compared to the mutator (which, for example, manipulates large Spark RDD arrays), the GC has much worse locality. Moreover, KMeans suffers much more from remote memory than TC due to significantly increased full-heap GC time. This is because KMeans uses a number of persisted RDDs (that are held in memory indefinitely). Although TC also persists RDDs, those RDDs are too large to be held in memory; as such, Spark releases them and reconstructs them when they are needed. This increases the amount of computation but reduces the GC effort under disaggregation. However, since memoization is an important and widely used optimization, it is not uncommon for data processing applications to hold large amounts of data in memory. As a result, these applications are expected to suffer from large-working-set GC as well.

These results call for a new managed runtime that can deliver good performance under disaggregated memory without requiring developers to be aware of and reason about the effects of disaggregation during development.

3 *Semeru* Heap and Allocator

This section discusses the design of *Semeru*'s memory abstraction. In order to support legacy applications developed for monolithic servers and to hide the complexity of data movement, we propose the *universal Java heap (UJH)* memory abstraction. We first describe this abstraction, and then discuss object allocation and management.

3.1 Universal Java Heap

The main process (*i.e.*, a JVM instance) running on the CPU server sees a large contiguous virtual address space, which we refer to as the universal Java heap. The application can access any part of the heap regardless of the physical locations. This contiguous address space is *partitioned* across memory servers, each of which provides physical memory that backs a disjoint region of the universal heap. The CPU server also has a small amount of memory, but this memory will serve as a software-managed, inclusive cache and hence not be dedicated to specific virtual addresses. Mutator (*i.e.*, application) threads run on the CPU server. When they access pages that are uncached on the CPU server, a page fault is triggered, and the paging system swaps pages that contain needed objects into the CPU server's local memory (cache). When the cache is full, selected pages are swapped out (evicted) to their corresponding memory servers, as determined by their virtual addresses.

Figure 2(a) provides an overview of the UJH. In addition to the main process running on the CPU server, *Semeru* also runs a *lightweight* JVM (LJVM) process on each participating memory server that performs tracing over local objects. This LJVM[3] is specially crafted to contain only the modules of object tracing and memory compaction, with support for RDMA-enabled communication with the CPU server. Due to its simplicity (*i.e.*, the modules of compiler, class loader, and runtime as well as much of the GC are all eliminated), the LJVM has a very short initialization time (*e.g.*, milliseconds) and low memory footprint (*e.g.*, megabytes of memory for tracing metadata). Hence, a memory server can easily run many LJVMs despite its weak compute (*i.e.*, each for a different CPU-server process).

When the LJVM starts, it aligns the starting address of its local heap with that of its corresponding address range in the UJH. As a result, each object has the same virtual address on the CPU and memory servers, enabling memory servers to trace their local objects without address translation. All physical memory required at each memory server is allocated when the LJVM is launched and pinned down during the entire execution of the program.

Coherency. This memory abstraction is similar in spirit to distributed shared memory (DSM) [66], which has been studied for decades. However, different from DSM, which needs to provide strong coherency between servers, *Semeru*'s coherency protocol is much simpler because memory servers, which collectively manage the address space, do not execute

[3] It is technically no longer a JVM since it does not execute Java programs.

any mutator code. The CPU server has access to the entire UJH, but each memory server can only access data in the address range it manages. In *Semeru*, each non-empty virtual page is in one of two high-level states, *cached* (in the CPU server) or *evicted* (to a memory server). When the CPU server accesses an *evicted* virtual page, it swaps the page data into its cache and changes the page's state to *cached*.

3.2 Allocation and Cache Management

Object allocation is performed at the CPU server. Allocation finds a virtual space that is large enough to accommodate the object being allocated. We adopt G1's region-based heap design where the heap is divided into *regions*, which are contiguous segments of virtual memory. The region-based design enables *modular tracing and reclamation* — each memory server hosts a set of regions; a memory server can trace any region it hosts independently of other regions, thereby enabling memory servers to perform tracing in parallel (while the CPU server executes the program). Modular tracing is enabled by using *remembered sets*, discussed shortly in §4.

When an object in a region is requested by the CPU server, the page(s) containing the object are swapped in. At this point, the region is *partially cached* and registered at the CPU server into an *active region list*. *Semeru* uses a simple LRU-based cache management algorithm to evict pages. The region is removed from this list whenever all its pages are evicted.

Upon an allocation request, the *Semeru* allocator finds the first region from this list that has enough space for the new object. If none of these regions can satisfy the request, *Semeru* creates a new region and allocates the object there. Allocation is based upon an efficient *bump pointer* algorithm [57], which places allocated objects contiguously and in allocation order. Bump pointer allocation maintains a position pointer for each region, pointing to the starting address of the free space. Bump pointer allocation maintains a position pointer for each region that points to the starting address of the region's free space. For each allocation, the pointer is simply "bumped up" by the size of the allocated object. Very large objects are allocated to a special heap area called the *humongous space*.

```
1 struct region {
2   uint64_t start;        // start address
3   uint64_t bp;           // bump pointer
4   uint64_t num_obj;      // total # objects
5   uint64_t cached_size; // size of pages in CPU cache
6   uint16_t survivals;    // # evacuations survived
7   remset* rem_set;       // remembered set (Section 4)
8   ...
9 }
```

Figure 3: A simplified definition for a region descriptor in *Semeru*.

The CPU server maintains, for all regions, their state *descriptors*. Each region descriptor is a `struct`, illustrated in Figure 3. Descriptors are used in both allocation and garbage collection. For example, `start` and `bp` are used for allocation; they can also be used to calculate the size of allocated objects. `survivals` indicates the total number of evacuation phases that the regions' objects have survived. It can be used, together with `num_obj`, to compute an *age* measurement for the region. `rem_set` is used as the tracing roots, which will be discussed shortly in §4.2.

Cache Management. *Semeru* employs a lazy write-back technique for allocations. Each allocated object stays in the CPU server's cache and *Semeru* does not write the object back to its corresponding memory server until the pages containing the object are evicted. For efficiency, only dirty pages are written back. Figure 2(b) shows the state machine of a virtual page. Each virtual page is initially in the `Init` state. Upon an object allocation on a page, the object is placed in the cache of the CPU server and its virtual page is marked as `Cached`, indicating that the object is currently being accessed by the CPU server. Evicted pages are swapped out to memory servers. Virtual pages freed by the GC are *unmapped* from their physical pages (their corresponding page table entries are *not* freed) and have their states reset to `Init`. This state machine is managed solely by the CPU server; memory servers do not run application code and hence do not need to know the state of each *page* (although they need to know the state of *regions* for tracing).

4 *Semeru* Distributed Garbage Collector

Semeru has a distributed GC that offloads tracing — the most memory-intensive operation in the GC (as it visits every live object) — to memory servers. Tracing is a task that fits well into the capabilities of a memory server with limited compute. That is, traversing an object graph by chasing pointers does not need strong compute, but benefits greatly from being close to data. In addition to memory-server tracing that runs continuously, *Semeru* periodically conducts a highly parallel stop-the-world (STW) collection phase to free cache space on the CPU server and reclaim memory on memory servers by evacuating live objects.

Design Overview. Although regions have been used in prior heap designs [36, 79], there are two unique challenges in using regions efficiently for disaggregated memory.

The first challenge is *how to enable modular tracing for regions.* Prior work such as Yak [79] builds a *remembered set* (remset) for each region that records references coming into objects in the region from other regions. These references, which are recorded into the set by instrumentation code called a *write barrier*, when the mutator executes each *object write of a non-null reference value*, can be used as additional *roots* to traverse the object graph for the region. However, none of the existing techniques consider a distributed scenario, where region tracing is done on memory servers, while their remsets are updated by mutator threads on the CPU server. We propose a new distributed design of the remset data structure to minimize the communication between the CPU and memory servers. Our remset design is discussed in §4.1.

The second challenge is *how to split the GC tasks between servers.* Our distributed GC has two types of collections:

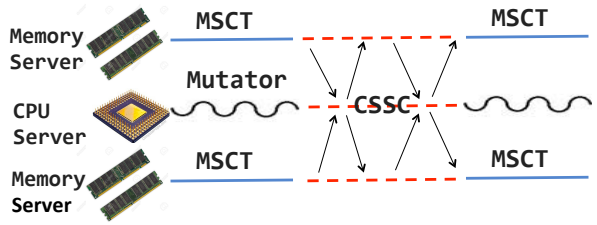


Figure 4: *Semeru* GC overview: the MSCT (on memory servers) traces evicted regions; the CSSC (coordinated between CPU and memory servers) traces cached regions and reclaims all regions.

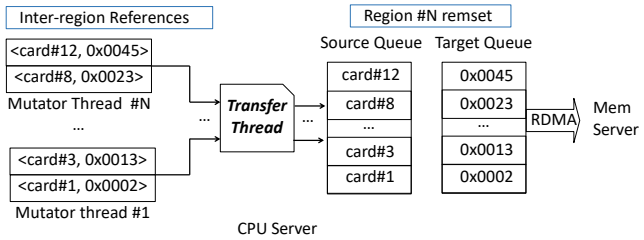


Figure 5: *Semeru*'s remset design; the source and target queues are implemented as bitmaps for space efficiency.

Memory Server Concurrent Tracing (MSCT, §4.2): Each memory server performs *intra-region* tracing over *regions for which most pages are evicted*, as a *continuous task*. Tracing runs *concurrently* on memory servers by leveraging their cheap but idle CPU resources. One can think of this as a background task that does not add any overhead to the application execution. The goal of MSCT is to compute a live object closure for each region at memory servers without interfering with the main execution at the CPU server. As a result, by the time a STW phase (*i.e.*, CSSC) runs, much of the tracing work is done, minimizing the STW pauses.

CPU Server Stop-The-World Collection (CSSC, §4.3): The CSSC is the main collection phase, coordinated between the CPU and memory servers to reclaim memory. During this phase, memory servers follow the per-region object closure computed during the MSCT to evacuate (*i.e.*, move out) live objects. Old regions are then reclaimed as a whole. Also during this phase, the CPU server traces and reclaims regions *for which most pages are cached*. Such regions are not traced by the MSCT. For evacuated objects, pointers pointing to them need to be updated in this phase as well.

Figure 4 shows an overview of these two types of collections. While the CPU server runs mutator threads, memory servers run the MSCT that continuously traces their hosted regions. When the CPU server stops the world and runs the CSSC, memory servers suspend the MSCT and coordinate with the CPU server to reclaim memory.

4.1 Design of the Remembered Set

The remset is a data structure that records, for each region, the references coming into the region. The design of the remset is much more complicated under a memory-disaggregated architecture due to the following two challenges. First, in a traditional setting, to represent an inter-region reference (*e.g.*, from field $o.f$ to object p), we only need its *source* location — the address of $o.f$. This is because p can be easily obtained by following the reference in $o.f$. However, in our setting, both $o.f$ and p need to be recorded for efficiency. This is because o and p can be on different servers and naïvely following the reference in $o.f$ can trigger a remote access.

The second challenge is that the remset of each region is updated by the write barrier executed on the CPU server, while the region may be traced by a memory server. As a result, the CPU server has to periodically send the remsets to memory servers for them to concurrently trace their regions. In addition, after memory servers evacuate objects, they need to send update addresses for the remsets back to the CPU server for it to update the sources of references (*e.g.*, $o.f$ may point to a moved object p).

Figure 5 shows our remset. To represent the source of a reference, we leverage OpenJDK's *card table*, which groups objects into fixed-sized buckets (*i.e.*, cards) and tracks which buckets contain references. A card's ID can be easily computed (*i.e.*, via a bit shift) from a memory address and yet we can enjoy the many space optimizations already implemented in OpenJDK (*e.g.*, for references on *hot* cards that contain references going to the same region [36], their sources need to be recorded only once). As such, each incoming reference is represented as a pair $\langle$`card`, `tgt`$\rangle$ where `card` is the (8-byte) index of the card representing the source location of the reference, and `tgt` is the (8-byte) address of the target object.

Shown on the left side of Figure 5 are inter-region references recorded by the write barrier of each mutator thread. To reduce synchronization costs, each mutator thread maintains a thread-local queue storing its own inter-region references. The CPU-server JVM runs a daemon (transfer) thread that periodically moves these references into the remsets of their corresponding regions (*i.e.*, determined by the target addresses). For each region, a pointer to its remset is saved in the region's descriptor (Figure 3), which can be used to retrieve the remset by the CPU server. When a reference is recorded in a remset, its `card` and `tgt` are decoupled and placed separately into a source and a target queue.

Target queues are sent (together with stack references) — during each CSSC via RDMA — to their corresponding memory servers, which use them as roots to compute a closure over live objects. Source queues stay on the CPU server and are used during each CSST to update references if their target objects are moved during evacuation. The benefit of using a transfer thread is that mutator threads simply dump inter-region references, while the work of separating sources and targets and deduplicating queues (based on a simple hash-based data structure) is done by the transfer thread, which does not incur overhead on the main (application) execution.

4.2 Memory Server Concurrent Tracing (MSCT)

The MSCT brings significant efficiency benefits because (1) tracing computation runs where data is located, avoiding

high swapping costs, and (2) tracing regions *concurrently* on multiple memory servers has zero impact on the execution of the main application on the CPU server.

The MSCT *continuously* traces regions (until the CSSC starts) in the order of a region's age (*i.e.*, the smaller the value of `survivals`, the younger a region) and the percentage of evicted pages. That is, younger regions with more evicted pages are traced earlier. This is because (1) younger regions are likely to contain more garbage (according to the generational hypothesis), and (2) evicted pages are not touched by the CPU server. Regions with a low ratio of evicted pages are *not* traced since cached objects may be frequently updated by the CPU server. Tracing such regions would be less profitable because these updates can change pointer structures frequently, making the tracing results stale.

Identifying Roots. There are two types of roots for the MSCT to trace a region: (1) objects referenced by stack variables and (2) cross-region references recorded in the region's remset. Both types of information come from the CPU server — during each CSSC (§4.3), the CPU server scans its stacks, identifies objects referenced by stack variables, and sends this information, together with each region's remset, to its corresponding memory server via RDMA.

Live Object Marking. The MSCT computes a closure of reachable objects in each region by traversing the object subgraph (within the region) from its roots. When live objects are traversed, we remember them in a per-region bitmap `live_bitmap` where each bit represents a contiguous range of 8 bytes (because the size of an object is always a multiple of 8 bytes), and the bit is set if these bytes host a live object. Furthermore, since live objects will be eventually evacuated, we compute a new address for a live object as soon as it is marked. The new address indicates where this object will be moved to during evacuation. New addresses are recorded in a *forward table* (*i.e.*, a key–value store) where keys are the indexes of the set bits in `live_bitmap` and values are the new addresses of the live objects represented by these bits.

Each new address is represented as an *offset*. At the start of the MSCT, it is unclear where these objects will be moved to (since evacuation will not be performed until a CSSC). As a result, rather than using absolute addresses, we use offsets to represent their relative locations. Their actual addresses can be easily computed using these offsets once the starting address of the destination space is determined.

Offset computation is in *traversal order*. For example, the first object reached in the graph traversal receives an offset 0; the offset for the second object is the size of the first object. This approach dictates that *objects that are contiguous in traversal will be relocated to contiguous space after evacuation*. Hence, the traversal order, which determines which objects will be *contiguously placed* after evacuation, is critical for improving data locality and prefetching effectiveness.

For instance, if the traversal algorithm uses DFS, *objects connected by pointers* will be relocated to contiguous memory (based on an observation that such objects are likely in the same logical data structure and hence accessed contiguously). As another example, if we use BFS to traverse the graph, *objects at the same level of a data structure* (such as elements of an array) will be relocated to contiguous memory; this can be useful for streaming applications that may do a quick linear scan of all such element objects (*i.e.*, BFS) rather than fully exploring each element (*i.e.*, DFS). To support these different heuristics, *Semeru* allows the user to customize the traversal algorithm for different workloads.

Tracing Correctness. There are two potential concerns in tracing safety. First, if a region has a cached page, can the memory server safely trace the region (given that the CPU server may update the cached page)? For example, if an update happens after tracing completes, would the tracing results still be valid? Second, the root information may be out of date when a region is traced because the CPU server may have updated certain inter-region references or stack variables since the previous CSSC (where roots are computed and sent). Is it safe to trace with such out-of-date roots?

The answer to both questions is that it is still valid for a memory server to trace a region over an *out-of-date* object graph. An important safety property is that *objects unreachable in any snapshot of the object graph will remain unreachable in any future snapshots* (*i.e.*, "once garbage, always garbage"). Thus the transitive closure may include dead objects (due to pointer changes the memory server is not aware of), but objects *not* in the closure are guaranteed to be dead (except for newly allocated objects, discussed next).

However, tracing using an out-of-date object graph may lead to two issues. First, the CPU server may allocate new objects into a region *after* the region is traced on a memory server. These new objects are missed by the closure computation. To solve this problem, we identify *all* objects that have been allocated into the region *since the last CSSC*; such objects are all marked live at the time the region is reclaimed in the next CSSC so that no live object is missed. Newly allocated objects can be identified by remembering the value of the bump pointer (`bp` in Figure 3) at the the last CSSC and comparing it with the current value of `bp` — the difference between them captures objects allocated since the last CSSC. Such handling is *conservative*, because some of the objects may be dead already but are still included in the closure.

The second issue is that some objects in the region may lose their references and become unreachable after tracing is done. These dead objects are still in the closure. For this issue, we take a *passive* approach by not doing anything — we simply let these dead objects stay in the closure and be moved during evacuation. These dead objects will be identified in that next MSCT and collected during the next CSSC. Essentially, we delay the collection of these objects by one CSSC cycle. Note that datacenter applications are often not resource strapped; hence, delaying memory reclamation by one GC cycle is a better choice than an aggressive alternative that retraces the

region before reclamation (which can increase the length of each CSSC pause).

Handling CPU Evictions. A significant challenge is that concurrent tracing of a region can potentially *race* with the CPU server evicting a page into the region. To complicate matters, memory servers are not aware of remote reads/writes due to *Semeru*'s use of one-sided RDMA (for efficiency). Although recent RDMA libraries (such as LITE [91]) provide rich synchronization support, our use of RDMA at the block layer has many specific needs that are not met by these libraries, which were developed for user-space applications.

To overcome this challenge, we develop a simple workaround: each memory server reserves the *first 4 bytes of each region* to store two tags $\langle dirty, ver \rangle$. The first 2 bytes encode a boolean *dirty* tag and the second 2 bytes encode an integer *version* tag. These two tags are updated by the CPU server both before and after evicting pages into a region, and checked by the memory server both before and after the region is traced. Figure 6 shows this logic.

```
|-- 16-bit Dirty tag   --
|-- 16-bit Version tag --|

1 atomic_write(<1, v1>);
2 evict_pages();
3 atomic_write(<0, v2>);
```

(a) CPU server eviction

```
4  <dirty, ver> = atomic_read();
5  if(!dirty) {
6    trace();
7    <dirty, ver1> = atomic_read();
8    if(ver != ver1) discard();
9  }
10 else skip();
```

(b) MSCT tracing

Figure 6: Detection of evictions at a memory server.

Before evicting pages, the CPU server assigns 1 to the dirty tag and a new version number v_1 to the version tag (Line 1). This 4-byte information is written atomically by the RDMA network interface controller (RNIC) into the target region. After eviction, the CPU server clears the dirty tag and writes another version number v_2 (Line 3). The memory server reads these 4 bytes atomically and checks the dirty tag (Line 4). If it is set, this indicates a potential eviction; the memory server skips this region and moves on to tracing the next region (Line 10). Otherwise, the region is traced (Line 6). After tracing, this metadata is retrieved again and the new version tag is compared with the pre-tracing version tag. A difference means that an eviction may have occurred and the tracing results are discarded (Line 8).

The algorithm is sufficient to catch all concurrent evictions. The correctness can be easily seen by reasoning about the following three cases. (1) If Line 1 comes before Line 4 (which comes before Line 3), tracing will not be performed. (2) If Line 1 comes after Line 4 but before Line 8, the version check at Line 8 will fail. (3) If Line 1 comes after Line 7, the eviction has no overlap with the tracing and thus the tracing results are legitimate.

This algorithm introduces overheads due to extra write-backs. However, by batching pages from the same region and employing InfiniSwap's gather/scatter, we manage to reduce this overhead to about 5%, which can be easily offset by the savings achieved by tracing objects on memory servers (see §6.4). Concurrent CPU-server reads are *allowed*. Similar to tracing out-of-date object graphs, fetching a page into the CPU server can potentially lead to new objects and pointer updates to the page. However, our aforementioned handling is sufficient to cope with such scenarios.

4.3 CPU Server Stop-The-World Collection (CSSC)

CSSC Overview. As the major collection effort, the CSSC runs when (1) the heap usage exceeds a threshold, *e.g.*, $N\%$ of the heap size, or (2) *Semeru* observes large amounts of swapping. The CPU server suspends all mutator threads and collaborates with memory servers to perform a collection. Our goal is to (1) reclaim cache memory at the CPU server and (2) provide a STW phase for memory servers to safely reclaim memory by evacuating live objects in the traced regions. Figure 7 overviews the CSSC protocol; edges represent communications of GC metadata between CPU and memory servers. The CSSC has four major tasks.

Task 1: The CPU server prepares information for memory servers to reclaim regions. Such information includes which regions to reclaim at each memory server (❶) and newly allocated objects for each region to be reclaimed (❷). As discussed in §4.2, newly allocated objects need to be marked live for safety and are identified by differencing the current value of bp and its old value (old_bp) captured in the last CSSC. This information is sent to memory servers (❷ → ⓾) before they reclaim regions. Before evacuation happens, each memory server must ensure that regions to be evacuated have all their pages evicted, to avoid inconsistency. To this end, the CPU server evicts all pages for each selected region (❶).

Task 2: Memory servers reclaim selected regions by moving out their live objects (⓾ – ⓮). For these regions, their tracing (*i.e.*, closure computation) is already performed during the MSCT, and hence, reclamation simply follows the closure to copy out live objects (*i.e.*, object evacuation) from old regions into new ones. Object evacuation is done using a region's forward table, which is computed in traversal order to improve locality, as discussed earlier in §4.2. Live objects from multiple old regions can be compacted into a new region to reduce fragmentation. Moreover, each memory server attempts to coalesce regions connected by pointers, again, to improve locality — if region A has references from region B, *Semeru* attempts to copy live objects from A and B into the same (new) region. The new addresses of these objects can be computed easily by adding their offsets from the forward tables onto the base addresses of their target spaces (which may be brand-new or half-filled regions).

Since objects are moved, their addresses have changed and hence pointers (stack variables or fields of other objects) referencing the objects must be updated. Pointer updates, however, must be done through the CPU server, because pointers can be scattered across the cache and other memory servers. Thus after reclaiming regions, each memory server

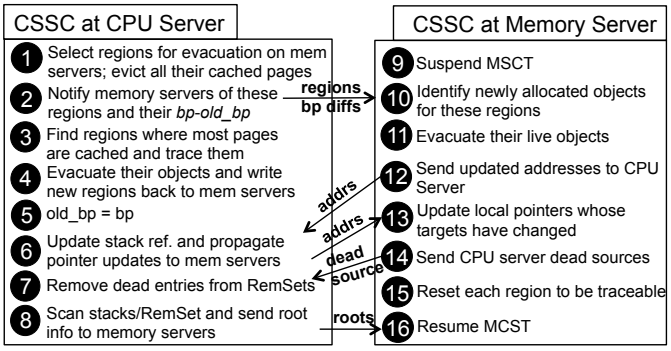


Figure 7: *Semeru*'s CSSC protocol: edges represent communications in the RDMA control path; $bp - old\_bp$ represents the difference between the current bp and the value of bp captured at the last CSSC.

sends the updated addresses of moved objects back to the CPU server (⓬). If a cached object references a moved object, the CPU server updates the pointer directly; the CPU server must also propagate these update requests to other memory servers (❻ → ⓭), which may host objects referencing moved objects.

Task 3: While memory servers reclaim their regions, the CPU server reclaims regions where most objects are cached. Since these regions have not been traced during the MSCT, the CPU server has to trace them to build the closure and then reclaim them using the same object evacuation algorithm (❸ and ❹). Unlike memory-server region reclamation, the CPU server has to additionally write new regions back to their respective memory servers after object evacuation to ensure consistency (❹). Next, the CPU server remembers the current value of bp into old_bp (❺) for use in the next CSSC.

Task 4: Since most dead objects have already been reclaimed, the CPU server scans the remsets to remove *dead entries* (❼). This is important since otherwise remsets can keep growing and dead entries would become memory leaks. Removing dead entries at the CPU server requires memory servers to provide information about which objects are dead (⓮ → ❼) because most regions are traced and reclaimed at memory servers. The CPU server then scans each reference in each region's remset and removes those references with dead targets. Finally, the CPU server scans its stacks and the updated remset of each region to compute new roots, which are sent to memory servers for the next round of MSCT (❽). Memory servers reset the metadata (*e.g.*, `live_map` and forward table) so that the next round of MSCT can trace each region from scratch (⓯ and ⓰).

Since each CSSC only collects selected regions, it may not reclaim enough memory for the application to run. In such rare cases (*e.g.*, one or two in our experiments with each Spark application), *Semeru* runs a full-heap scan (*i.e.*, the same as a regular full-heap GC in G1), which brings all objects into the cache for tracing and collection. Since CSSC relies on remset-based modular tracing, it cannot reclaim dead objects that are (1) in different regions and (2) form cycles. Such objects have to be reclaimed at a full-heap GC.

5 The *Semeru* Swap System

We build *Semeru*'s swap system by piggybacking on Mellanox's NVMe-oF implementation [1]. This section briefly describes our modifications. During booting, the CPU server sends JVM metadata (such as metadata of loaded classes) in its *native heap* to memory servers, which use such information to launch LJVMs. On each memory server, the LJVM receives these native objects and reconstructs their virtual tables for function calls to execute correctly on these objects.

Block Layer. We modify NVMe-oF's block layer to add support for remote memory management. The remote physical memory that backs the Java heap on all memory servers is registered as a whole as an RDMA buffer and pinned down throughout the execution. As a simple optimization, we remove block-layer staging queues and merge several block I/O (BIO) requests into a single I/O request, turning them directly into RDMA messages.

Merging BIOs enables the use of InfiniBand's gather-scatter for data transfer. For each BIO request generated by the block layer, it often contains multiple physical pages to be transferred to a memory server. These physical pages are not necessarily contiguous. One optimization here is instead of generating multiple RDMA messages separately for these physical pages, we amortize per-message overhead by leveraging the scatter-gather technique so that these pages can be processed using a single RDMA message. We also develop thread-local RDMA message pools so that multiple threads can perform their own RDMA message creation and initialization without needing synchronization.

RDMA Management. All communications between the CPU and memory servers are through reliable one-sided RDMA. We distinguish these communications based on data types: (1) page fetching and evictions, which dominate the communications, go through a *data path* inside the kernel (to provide transparency to applications); (2) signals and GC information (*e.g.*, all messages in Figure 7), are passed through a *control path* implemented as a user-space library for efficiency. A user-space implementation benefits from efficiency from raw RDMA (*e.g.*, no overhead from system calls); since the control path does not overlap with the data path and transfers small amounts of information (*i.e.*, only inside each CSSC), our implementation can deliver good performance for both control and data paths.

6 Evaluation

To implement *Semeru*, we wrote/modified 58,464 lines of (non-comment) C/C++ code, including 43,838 lines for the LJVM (based upon OpenJDK version 12.0.2) on memory servers, 7,406 lines for the CPU-server JVM, and 7,220 lines for the Linux kernel (4.11-rc8). Our kernel support contains 4,424 lines of C code for the paging system and RDMA

management (based upon NVMe-oF), and 2,796 lines for the modified block layer and memory management part as well as new system calls.

Setup and Methodology. We ran *Semeru* in a cluster with one CPU server and three memory servers. Each server has two Xeon(R) CPU E5-2640 v3 processors, 128GB memory, one 200GB SSD, and one 40 Gbps Mellanox ConnectX-3 InfiniBand network adapter. Servers are connected by one Mellanox 100 Gbps InfiniBand switch. To emulate the weak compute of memory servers, we let the LJVM on each memory server use only one core. All our experiments used a 32GB heap, 512MB regions, and 4K pages. The default swap prefetching mechanism in Linux was used.

Unfortunately, we were only able to gain exclusive use of a small cluster with four machines when evaluating *Semeru*. Despite running on this small cluster, our experiments used large-scale applications involving multiple memory servers, representing a real-world use of *Semeru*. Adding more memory servers would not change the results because (1) memory servers perform modular collection — they do not communicate with each other and hence not have scalability issues; and (2) the CPU server only communicates with memory servers during each CSSC — more memory servers would only increase the control-path communication, which is minimal. Adding CPU servers and running more processes would increase the amount of tracing work on each memory server. However, as shown in §6.3, tracing for a large Spark application can only utilize 13% of each memory server's compute — one single core on each server can support simultaneous tracing for ∼8 Spark applications.

Name	Dataset	Size
GraphX-ConnectedComponents (**GCC**) GraphX-PageRank (**GPR**)	Wikipedia English [5]	2GB
Naïve-PageRank (**NPR**)	Wikipedia Polish [5]	1GB
Naïve TriangleCounting (**NTC**)	Synthetic 2.5K points 10K edges	1GB
MLlib-Bayes Classifiers (**MBC**)	KDD 2012 [4]	5GB

Table 1: Description of five Spark programs.

Name	Dataset	Size
Word Count (**FWC**)	Wikipedia English [5]	2GB
KMeans (**KMS**)		
Connected Components (**FCC**)		

Table 2: Description of three Flink batch-processing programs.

We evaluated *Semeru* with two widely deployed data analytics systems: Apache Spark (3.0.0) and Apache Flink (1.10.1). Spark was executed under Hadoop 3.2.1 and Scala 2.12.11, using a set of five programs (listed in Table 1): PageRank (GPR) and ConnectedComponents (GCC) from the GraphX [48] libraries, as well as Bayes Classifier (MBC) from the MLlib libraries. We also included naïve PageRank (NPR) and naïve TriangleCounting (NTC), implemented directly atop Spark.

Flink also ran on top of Hadoop version 3.2.1. Flink has both streaming and batch-processing models. In this experiment, we focused on the batch-processing model, in particular, Map/Reduce programs. The programs and their datasets are summarized in Table 2. These programs are selected based on their popularity and usefulness, covering a spectrum of text analytics, graph analytics, and machine learning tasks.

6.1 Overall *Semeru* Performance

We compared *Semeru* and the original OpenJDK 12 that runs the G1 GC — the default GC in the JVM since OpenJDK 9. G1 is a concurrent GC that runs concurrent tracing as the mutator thread executes and stops the world for memory reclamation. G1 is designed for short latency (*i.e.*, GC pauses) at the cost of reduced throughput (*i.e.*, concurrent tracing slows down the mutator as it competes resources with the mutator). We have tested other GCs as well and found that G1 consistently outperforms all others in latency.

We ran G1 with two swap mechanisms: a local RAMDisk and NVMe-oF, which connects the CPU server to remote memory on the three memory servers. To use NVMe-oF, we configured remote memory as remote RAMDisks, which host data objects without supporting memory management. *Semeru* ran on our own swap system built on top of NVMe-oF with added support for the remote heap and memory management. Each memory server hosts around one-third of the 32GB Java heap. There are three cache configurations: 100%, 50%, and 25%. The 100% configuration is our baseline, which represents the original OpenJDK's performance without any swapping.

Running Time. Figure 8 shows performance comparison between these systems, for our eight programs, under the three cache configurations. There is only one bar under the 100% cache configuration, representing the original performance of G1 that does not perform swapping.

	50% Cache			25% Cache		
System	**Mutator**	**GC**	**All**	**Mutator**	**GC**	**All**
G1-RD	1.82×	2.79×	1.87×	3.16×	4.59×	3.23×
G1-NVMe	2.00×	4.44×	2.24×	3.85×	14.13×	4.58×
Semeru	**1.06**×	**1.42**×	**1.08**×	**1.22**×	**2.67**×	**1.32**×

Table 3: Overhead summary: overheads are calculated using the G1 performance under the 100% cache configuration as the baseline.

Table 3 summarizes the time overheads incurred by memory disaggregation on these systems. The baseline used to calculate these overheads is the G1 performance under the 100% cache ratio (without any kernel and JVM modification). On average, G1 has 1.87× and 2.24× end-to-end overhead under RAMDisk and NVMe-oF, respectively, for the 50% cache configuration. When the cache ratio reduces to 25%, these overheads increase to 3.23× and 4.58×, respectively. By offloading tracing and evacuation to memory servers and improving the locality for the mutator threads, *Semeru* reduces these overheads, by **3.23 times overall**, to **1.08**× and **1.32**× for the two cache ratios, respectively.

Our first observation here is that disaggregation incurs a much higher overhead on GC than the mutator for Spark

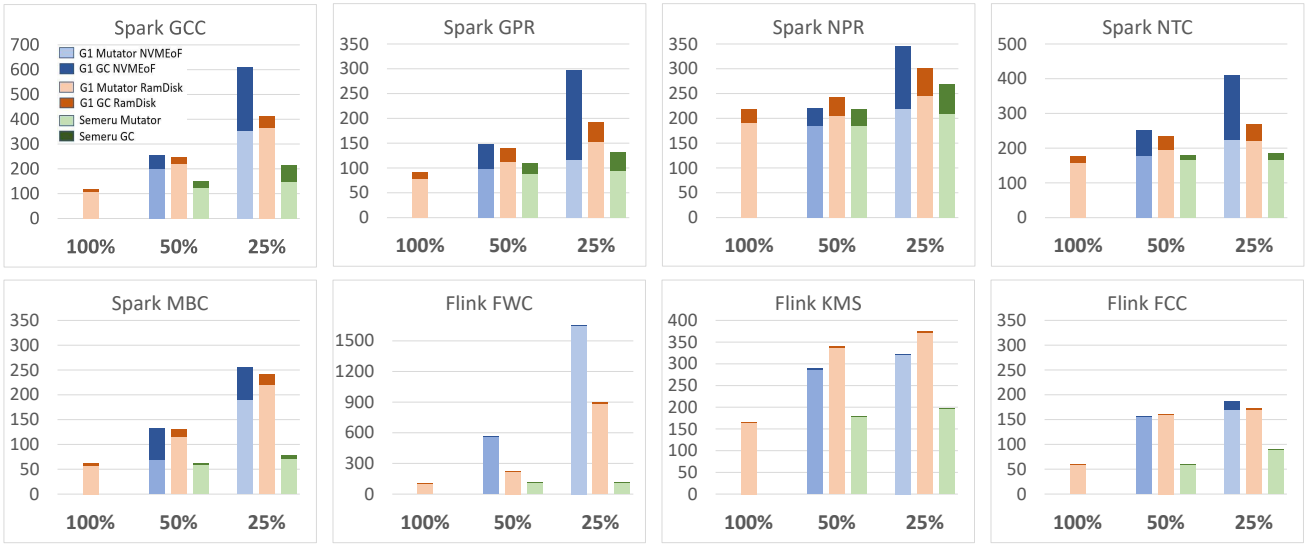


Figure 8: Performance comparisons between G1 under NVMe-oF (left bar of each group), G1 under RAMDisk (middle bar), and Semeru (right bar) for three cache configurations: 100%, 50%, and 25%; each bar is broken down into mutator (bottom) and GC (top) time (second).

applications, and it is consistent with our motivating data reported in §2. This is because GC algorithms inherently do not possess good locality and, as a result, pay a higher penalty for remote memory fetching than the mutator. This overhead grows significantly when the cache size decreases. It is also easy to see that accessing remote memory (via NVMe-oF) incurs a higher overhead than accessing the local RAMDisk.

The second observation is that for Flink, which has much less GC than Spark, *Semeru* can still considerably improve its performance. An inspection found that Flink stores data in the serialized form and implements operators that can process data without creating objects for them. Flink allocates long-lived data items directly in native memory and/or reserved space in the old generation. Nevertheless, *Semeru*'s optimizations are still effective. This is because the G1 GC uses a disaggregation-agnostic policy to dynamically tune the size of young generation. Since most objects in Flink die in the young generation, the pause time of each young GC is extremely short (*e.g.*, less than 10 ms) and always meets G1's pause-time target. As such, G1 keeps increasing the young generation size to reduce the GC frequency, making the young generation quickly reach the size of the CPU cache.

However, the problem here is the young generation contains large amounts of garbage, cached on the CPU server, leaving little cache space for long-lived data. This causes hot, long-lived data (*e.g.*, in native memory) to be frequently swapped in and out. In contrast, under *Semeru*'s region design, a CSSC is triggered when *Semeru* observes frequent swapping. The CSSC reclaims garbage and compacts regions, freeing up cache space for accommodating other hot data.

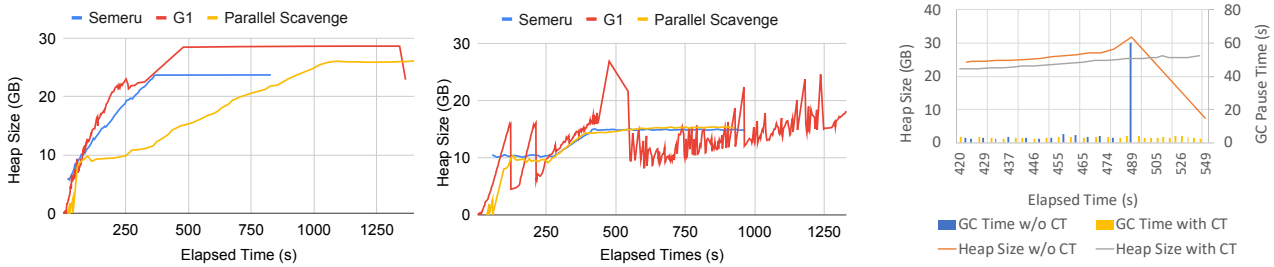
The third observation is that applications have different levels of tolerance to fetching latency. For example, GCC and GPR have an exceedingly high GC overhead because they create large RDDs and *persist* them in memory. These RDDs and their elements quickly become old and get promoted to the *old generation*. G1 cannot reclaim much memory in nursery GCs and, as such, most GCs scan the entire heap, requiring many remote fetches. For other applications such as Spark NPR, their GC performance is not as significantly degraded because their executions generate many temporary objects that die young (rather than old objects) — when a nursery GC runs, most young objects are garbage cached locally on the CPU server, and hence, they can be easily reclaimed without triggering many remote fetches.

System	50% Cache			25% Cache		
	Mutator	**GC**	**All**	**Mutator**	**GC**	**All**
G1-RD	1.73×	2.31×	1.75×	2.65×	2.35×	2.56×
G1-NVMe	1.91×	4.20×	2.10×	3.31×	5.61×	3.69×

Table 4: Summary of performance improvements achieved by *Semeru*: improvements are computed with $\frac{a}{b}$ where a is the (mutator, GC, or end-to-end) time under a system and cache configuration, and b is *Semeru*'s time under the same configuration.

To make *Semeru*'s improvements clear, Table 4 reports detailed improvement ratios under each configuration. It is easy to see that *Semeru* improves the performance of both the mutator and GC. On the mutator side, *Semeru* eliminates G1's concurrent marking — which runs on the CPU server in parallel with application execution, competing for resources with mutator threads and polluting the cache — and dynamically improves locality (discussed in §4.3) by relocating objects likely to be accessed to contiguous memory. On the GC side, *Semeru* significantly reduces pause time by letting memory servers perform tracing and evacuation, all of which used to be done on the CPU server.

Memory. To understand *Semeru*'s ability to reclaim memory, we collected post-GC memory footprints for Spark NPR and Spark KMS under three GCs: *Semeru*, G1, and Parallel



(a) Memory: NPR, 50% cache (b) Memory: KMS, 25% cache (c) Mem/time: NPR, w/ and w/o cont. tracing

Figure 9: Memory footprints under *Semeru*, G1, and Parallel Scavenge for NPR (a) and KMS (b); (c) shows the memory footprint and GC pause time with and without continuous tracing for NPR.

Scavenge (PS). PS is a non-concurrent GC designed for high throughput. We added PS because it often can reclaim more memory at each GC than G1 at the cost of higher latency. PS's strong memory reclamation capabilities are clearly seen in Figure 9(a) where PS has the lowest memory footprint throughout the execution. *Semeru* outperforms G1 — G1 uses concurrent tracing to estimate a garbage ratio for each region; with this information, when each STW phase runs, the GC can selectively reclaim regions with the highest garbage ratios. Under memory disaggregation, however, concurrent tracing runs slowly due to frequent remote fetches. It cannot finish tracing the heap at the time a STW starts; as a result, garbage ratios are not available for most regions.

As a result, at each STW phase, there is not much information about which regions have the most garbage, and thus, the GC selects arbitrary regions to collect. Many such regions do not have much garbage, which explains why G1 reclaims less memory than *Semeru* and PS. Note that *Semeru* does not suffer from this problem because tracing is done locally on memory servers; hence, it runs efficiently and can trace many regions between two consecutive CSSCs.

Figure 9(b) shows the memory footprint for Spark KMS running under the 25% cache configuration. In this case, *Semeru*'s collection performance is close to that of PS — for both of them, the program's memory consumption becomes stabilized after about 400 seconds. Under G1, however, the memory footprint fluctuates, again due to the (semi-random) selection of regions to collect. If regions with large garbage ratios happen to be in the cache, G1 is able to quickly identify them during concurrent marking and collect them in a subsequent STW phase. However, if they are remotely resident on memory servers, G1 would lack sufficient information in a STW phase to collect the right regions.

6.2 Effectiveness of Continuous Tracing

To understand the usefulness of continuous tracing on memory servers, we compared *Semeru* with a variant that does not perform continuous tracing but rather traces regions in each CSSC. In this variant, tracing is still done on memory servers but combined with other memory management tasks such as object evacuation in each STW phase. Without continuous tracing, which uses idle resources on memory servers to trace local regions, *Semeru* suffers from the same problem as G1 — when a CSSC runs, *Semeru* does not know which regions have the most garbage and thus should be reclamation targets. To minimize the GC latency, each CSSC has to be extremely short, leaving memory servers insufficient time to trace many regions. As a result, memory servers can only trace and reclaim regions based on their age without the more useful information of their garbage ratio.

To illustrate this problem, Figure 9(c) shows the post-GC memory footprint (*i.e.*, y-axis on the left) and the pause time of each CSSC (*i.e.*, y-axis on the right). The two lines represent the memory footprints of *Semeru* with and without continuous tracing while the short bars report the GC pauses. We make two important observations here. First, *Semeru* with continuous tracing consistently reclaims more memory than the version without continuous tracing, because it knows the right regions to reclaim in each CSSC. Second, since the version without continuous tracing cannot reclaim enough memory, it triggers a full-heap scan at the 484th second, which is extremely time consuming (*i.e.*, 65 seconds).

A modern generational GC achieves its efficiency by scanning only the young nursery generation in most of its GC runs. As soon as it needs to scan the entire heap, its performance degrades significantly. This is especially the case with memory disaggregation where a full-heap scan fetches most objects from memory servers to the CPU server, incurring an extremely long pause, as shown in the figure. The full-heap GC reclaims much space and reduces memory consumption.

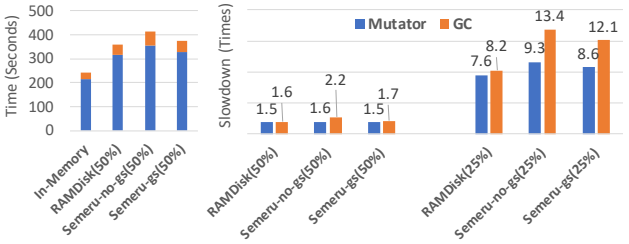
In contrast, with continuous tracing, *Semeru* does not encounter any full-heap GC throughout the execution. Although it does not reclaim as much memory as a full-heap GC, it avoids long pauses and yet is still able to give the application enough memory to run.

6.3 Tracing Performance

Memory servers are expected to possess weak compute power. To understand how tracing performs under different levels of compute, we used one single core on each memory and varied its frequency with DVFS. Table 5 summarizes the impact of each frequency on the tracing performance, GC and mutator performance, and end-to-end performance of NPR. We also obtained the same measurements when tracing is performed on the CPU server with a dedicated core. As shown, even with a single core at 1.2GHz, tracing on memory servers still yields

Configuration	Tracing Performance				Overall Performance		
	Thruput	CUtil	AT	AIT	GC	Mutator	Overall
(Memory Server) single core, 1.2 GHz	418.3 MB/s	29.0%	6.5 secs	4.6 secs	59.4 secs	180.2 secs	239.6 secs
(Memory Server) single core, 2.6 GHz	922.2 MB/s	12.4%	5.7 secs	5.0 secs	59.3 secs	173.9 secs	233.2 secs
(CPU Server) single core, 2.6 GHZ, dedicated to GC	93.9 MB/s	N/A	38.8 secs	N/A	126.0 secs	218.9 secs	344.9 secs

Table 5: Performance of NPR when tracing is performed under different core frequencies at memory servers: reported are the configurations (**Configuration**) of memory-server cores, tracing throughput (**Thruput**), memory-server CPU utilization (**CUtil**), average time between two consecutive CSSCs (**AT**), average idle CPU time between two consecutive CSSCs (**AIT**), total GC (**GC**) and mutator time (**Mutator**), and end-to-end run time (**Overall**).



(a) (b)

Figure 10: Comparisons between *Semeru*'s swap system and local RAMDisk: (a) shows Spark running times when the size of the cache is 50% of the heap size; the first bar reports performance of the baseline (cache ratio = 100%); (b) shows normalized performance (*i.e.*, slowdowns) for the two cache configurations (50% and 25%).

a throughput **4.5**× higher than doing so on the CPU server with a dedicated 2.6GHz core. This is easy to understand: the bottleneck of a memory-disaggregated system is at (1) the poor locality, which triggers many on-demand swaps, and (2) racing for network resource between the mutator and GC threads, *not* the lack of compute power.

Another important observation is on the low CPU utilization on memory servers. Even with a 1.2GHz core, continuous tracing between consecutive CSSCs has only 29% CPU utilization — this is because (1) tracing only follows pointers, (2) dead objects are not traced and hence, for each region, only a small fraction needs to be traced, and (3) not all regions need to be traced (*i.e.*, those with a high rate of cached objects are not traced). These results demonstrate that supporting multiple processes, with weak compute on memory servers, should not be a concern.

6.4 Swap Performance

To evaluate our swap system's performance, we turned off the *Semeru* runtime (*i.e.*, all memory management tasks on memory servers) and ran the original G1 GC on top of our swap system. We tried to run InfiniSwap [49], but its executions were frequently stuck, even on native programs. This subsection focuses on comparisons of swap performance between local RAMDisk and *Semeru*'s swap system (with and without using InfiniSwap's gather/scatter).

The results of Spark NPR are reported in Figure 10. We used two cache configurations: 50% and 25%. Figure 10(a) shows actual running times when the cache ratio is 50% between four versions of the system: in-memory (*i.e.*, cache ratio is 100%), RAMDisk, *Semeru*-no-gs (*i.e.*, gather/scatter is not used), and *Semeru*-gs (which uses gather/scatter). For ease of comparison, Figure 10(b) shows normalized times.

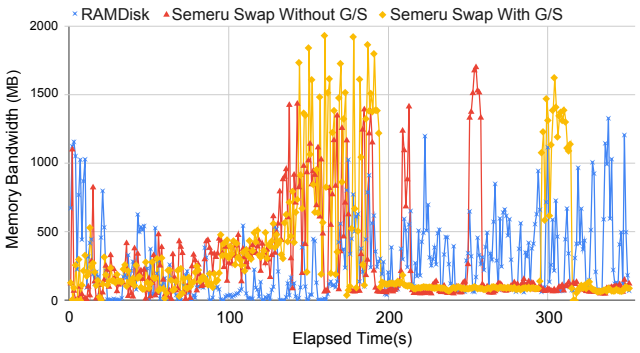


Figure 11: A comparison of the combined swap read/write throughput between *Semeru*-gs, *Semeru*-no-gs, and RAMDisk.

Under the 50% cache configuration, using RAMDisk as the swap partition incurs a 1.5× and 1.6× overhead in the mutator and GC, respectively, compared with the in-memory baseline. *Semeru*-no-gs increases the overheads to 1.6× and 2.2×. Merging BIO requests and using gather/scatter brings the overheads down to 1.5× and 1.7×, which are on par with those of the RAMDisk. Similar observations can be made for the 25% cache rate. Across all programs, gather/scatter improves the swap performance overall by 14%.

Figure 11 compares the read/write throughput between *Semeru*-gs, *Semeru*-no-gs, and RAMDisk when Spark LRG is executed under the 25% cache configuration. As shown, gather/scatter helps *Semeru* achieve a higher peak read/write bandwidth than *Semeru*-no-gs (especially when pages contiguously swapped come from / go to the same region).

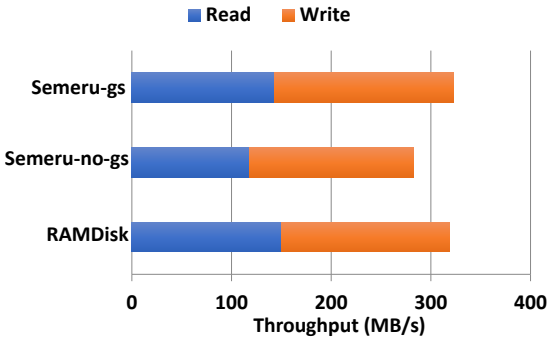


Figure 12: Average read/write throughput.

A comparison on the average read/write throughput between the three systems is shown in Figure 12. *Semeru*-gs's

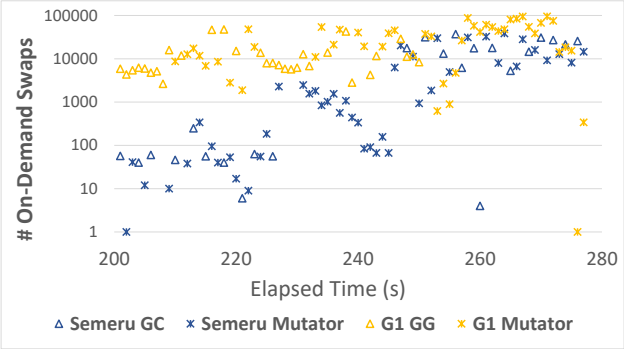


Figure 13: Numbers of on-demand swap-ins between G1 and *Semeru* under the 25% cache configuration for Spark MBC.

overall read/write throughput is 13% higher than that of *Semeru*-no-gs and is on par with that of RAMDisk. Clearly, additional gains can be obtained by merging BIO requests and using gather-scatter.

6.5 Locality Improvement

To understand how *Semeru* improves locality for application execution, we measured the number of on-demand swap-ins performed by the swap system under G1 and *Semeru* when Spark MBC was executed with a 25% cache ratio. Figure 13 reports how such numbers change as the execution progresses for both the mutator and GC. Both *Semeru*-muator and *Semeru*-GC need significantly fewer on-demand swap-ins due to improved locality. On average, *Semeru* reduces the number of on-demand swap-ins by **8.76**×. Note that both G1 and *Semeru* ran under the default swap prefetcher in Linux, which relies on the pages swapped in during the last two page faults: if they are contiguous, Linux continues to bring in several contiguous pages into the page cache; otherwise, it assumes that there are no patterns and reduces or stops prefetching. Despite the recent development of more advanced prefetchers (such as Leap [71]) for remote memory, *Semeru* already performs well under the default prefetcher in Linux. We expect it to continue to work well when other prefetchers are used. The average ratio between the sizes of data swapped in the data and control path is 29.8 across the programs.

7 Related Work

Resource Disaggregation. Due to rapid technological advances in network controllers, it has become practical to reorganize resources into disaggregated clusters [21, 29, 45, 51]. A disaggregated cluster can increase the hardware resource utilization and has the potential to overcome fundamental hardware limits, such as the critical "memory capacity wall" [9, 13, 17, 58, 67, 68, 95]. A good number of systems have been developed in the past to take advantage of this architecture [7, 35, 41, 42, 44, 54, 62]. However, almost all of them treat remote memory as fast storage. When the network connection only has microseconds of latency and hundreds of gigabits of bandwidth [55, 72], applications can suffer from significant delays in memory access. Despite many optimizations [7, 11, 49, 84, 87–89] developed to reduce this latency, they all focus on low-level system stacks and do not consider run-time characteristics of programs. They do not work well for managed cloud applications such as [6, 14, 15, 24–26, 31, 32, 50, 56, 75, 76, 81, 82, 92, 102–104, 106]. *Semeru* co-optimizes the runtime and the swap system, unlocking opportunities unseen by existing techniques.

Garbage Collection for Modern Systems. GC is a decades-old topic. In order to meet the requirements of low latency and high throughput, many concurrent GC algorithms have been proposed, including the Garbage-First (G1) GC [36], Compressor [59], ZGC [2], the Shenandoah GC [43], Azul's pauseless GC [34], and C4 [90], as well as several real-time GCs [18, 19]. These GC algorithms can run in the background with short pauses for mutator threads. However, none of them can work directly in the resource-disaggregated environment, which has a unique resource profile — data are all located on memory servers, the CPU server has a small cache, and memory servers have weak compute.

Efficiently using memory is important especially for applications running on the cloud [40]. Yak [79] is a region-based GC developed for such applications. Taurus [70] coordinates GC efforts in a distributed setting for cloud systems. Facade [80] uses region-based memory management to reduce GC costs for Big Data applications. Gerenuk [78] develops a compiler analysis and runtime system that enable native representation of data for managed analytics systems such as Spark and Hadoop. Espresso [99] and Panthera [95] are designed for systems with non-volatile memory. Platinum [98] is a GC that aims to reduce tail latency for interactive applications. NUMAGiC [47] is a GC developed for the NUMA architecture. However, NUMAGiC assumes that NUMA nodes are completely symmetric (with the same CPU, the same amount of local memory, and the same GC algorithm) — which is not the case for disaggregated clusters. DMOS [53] is a distributed GC algorithm that has not been implemented and whose performance in a real-world setting is unclear.

8 Conclusions

Semeru is a managed runtime designed for efficiently running managed applications with disaggregated memory. It achieves superior efficiency via a co-design of the runtime and swap system as well as careful coordination of different GC tasks.

Acknowledgments

We thank the OSDI reviewers for their valuable and thorough comments. We are grateful to our shepherd Yiying Zhang for her feedback, helping us improve the paper substantially. This work is supported by NSF grants CCF-1253703, CCF-1629126, CNS-1703598, CCF-1723773, CNS-1763172, CCF-1764077, CNS-1907352, CNS-1901510, CNS-1943621, CNS-2007737, CNS-2006437, and ONR grants N00014-16-1-2913 and N00014-18-1-2037, and a grant from the Alexander von Humboldt Foundation.

A Artifact Appendix

A.1 Artifact Summary

Semeru is a managed runtime built for a memory-disaggregated cluster where each managed application uses one CPU server and multiple memory servers. When launched on *Semeru*, the process runs its application code (mutator) on the CPU server, and the garbage collector on both the CPU server and memory servers in a coordinated manner. Due to task offloading and moving computation close to data, *Semeru* significantly improves the locality for both the mutator and GC and, hence, the end-to-end performance of the application.

A.2 Artifact Check-list

- **Hardware: Intel servers with InfiniBand**
- **Run-time environment: OpenJDK 12.02, Linux-4.11-rc8, CentOS 7.5(7.6) with MLNX-OFED 4.3(4.5)**
- **Public link: https://github.com/uclasystem/Semeru**
- **Code licenses: The GNU General Public License (GPL)**

A.3 Description

A.3.1 *Semeru*'s Codebase

Semeru contains the following three components:

- the Linux kernel, which includes a modified swap system, block layer and a RDMA module,
- the CPU-server Java Virtual Machine (JVM),
- the Memory-server lightweight Java Virtual Machine (LJVM).

These three components and their relationships are illustrated in Figure 14.

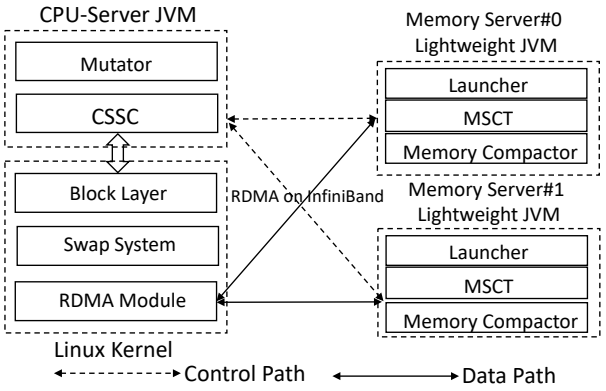


Figure 14: Overview of *Semeru*'s codebase.

A.3.2 Deploying *Semeru*

To build *Semeru*, the first step is to download its source code:

```
git clone
git@github.com:uclasystem/Semeru.git
```

When deploying Semeru, install the three components in the following order: the kernel on the CPU server, the *Semeru* JVM on the CPU server, and the LJVM on each memory server. Finally, connect the CPU server with memory servers before running applications.

Kernel Installation. We first discuss how to build and install the kernel.

- Modify `grub` and set `transparent_hugepage` to `madvise`:

```
sudo vim /etc/default/grub
+ transparent_hugepage=madvise
```

- Install the kernel and restart the machine:

```
cd Semeru/Linux-4.11-rc8
sudo ./build_kernel.sh build
sudo ./build_kernel.sh install
```

- Build the *Semeru* RDMA module:

```
# Add the IP of each memory server into
# Semeru/linux-4.11-rc8/include/
# linux/swap_global_struct.h
# e.g., the Infiniband IPs of the 2 memory servers
# are 10.0.0.2 and 10.0.0.4.
char* mem_server_ip[][] = {"10.0.0.2",
"10.0.0.4"};
uint16_t mem_server_port = 9400;
# Then build the Semeru RDMA module
make
```

Install the CPU-Server JVM. We next discuss the steps to build and install the CPU-server JVM.

- Download Oracle JDK 12 to build Semeru JVM:

```
# Assume jdk 12.02 is under path
# ${home_dir}/jdk12.0.2
# Or change the path in shell script
# Semeru/CPU-Server/build_cpu_server.sh
boot_jdk="${home_dir}/jdk12.0.2"
```

- Build the CPU-server JVM:

```
# ${build_mode} can be one of the three modes:
# slowdebug, fastdebug, or release.
# We recommend fastdebug to debug the JVM code
# and release to test the performance.
# Please make sure both the CPU server and
# memory servers use the same build mode.
cd Semeru/CPU-Server/
./build_cpu_server.sh ${build_mode}
./build_cpu_server.sh build
# Take fastdebug mode as example — the compiled
# JVM will be in:
# Semeru/CPU-Server/build
# /linuxx86_64serverfastdebug/jdk
```

Install the Memory-Server LJVM. The next step is to install the LJVM on each memory server.

- Download OpenJDK 12 and build the LJVM:

```
# Assume OpenJDK12 is under the path
#${home_dir}/jdk-12.0.2
# Or you can change the path in the script
# Semeru/Memory-Server/build_mem_server.sh
boot_jdk="${home_dir}/jdk-12.0.2"
```

- Change the IP addresses:

```
# E.g., mem-server #0's IP is 10.0.0.2, ID is 0.
# Change the IP address and ID in file:
# Semeru/Memory-Server/src/hotspot/share/
# utilities/globalDefinitions.hpp
#@Mem-server #0
#define NUM_OF_MEMORY_SERVER 2
#define CUR_MEMORY_SERVER_ID 0
static const char cur_mem_server_ip[] =
"10.0.0.2";
static const char cur_mem_server_port[]
= "9400";
```

- Build and install the LJVM:

```
# Use the same ${build_mode} as the CPU-server
# JVM.
cd Semeru/CPU-Server/
./build_memory_server.sh ${build_mode}
./build_memory_server.sh build
./build_memory_server.sh install
# The compiled Java home will be installed under:
# {home_dir}/jdk12u-self-build/jvm/
# openjdk-12.0.2-internal
# Set JAVA_HOME to point to this folder.
```

A.3.3 Running Applications

To run applications, we first need to connect the CPU server with memory servers. Next, we mount the remote memory pools as a swap partition on the CPU server. When the application uses more memory than the limit set by `cgroup`, its data will be swapped out to the remote memory via RDMA.

- Launch memory servers:

```
# Use the shell script to run each memory server.
#${execution_mode} can be execution or gdb.
#@Each memory server
cd Semeru/ShellScrip
run_rmem_server_with_rdma_service.sh
Case1 ${execution_mode}
```

- Connect the CPU server with memory servers:

```
#@CPU server
cd Semeru/ShellScript/
install_semeru_module.sh semeru
# To close the swap partition, do the following:
#@CPU server
cd Semeru/ShellScript/
install_semeru_module.sh close_semeru
```

- Set a cache size limit for an application:

```
# E.g., Create a cgroup with 10GB memory limita-
tion.
#@CPU server
cd Semeru/ShellScript
cgroupv1_manage.sh create 10g
```

- Add a Spark executor into the created `cgroup`:

```
# Add a Spark worker into the cgroup, memctl.
# Its sub-process, executor, falls into the same cgroup.
# Modify the function start_instance under
# Spark/sbin/start-slave.sh
#@CPU server
cgexec -sticky -g memory:memctl
"${SPARK_HOME}/sbin" /sparkdaemon.sh
start $CLASS $WORKER_NUM -webui-port
"$WEBUI_PORT" $PORT_FLAG $PORT_NUM
$MASTER "$@"
```

- Launch a Spark application:

  Some *Semeru* JVM options need to be added for both CPU-server JVM and LVJMs. CPU-server JVM and memory server LJVMs should use the value for the same JVM option.

```
# E.g., under the configuration of 25% local memmory
# 512MB Java heap Region
#@CPU server
-XX:+SemeruEnableMemPool
-XX:EnableBitmap -XX:-UseCompressedOops
-Xnoclassgc -XX:G1HeapRegionSize=512M
-XX:MetaspaceSize=0x10000000
-XX:SemeruLocalCachePercent=25
#@Each memory server
# ${MemSize}: the memory size of current memory
server
# ${ConcThread}: the number of concurrent threads
-XX:SemeruEnableMemPool
-XX:-UseCompressedOops
-XX:SemeruMemPoolMaxSize=${MemSize}
-XX:SemeruMemPoolInitialSize=${MemSize}
-XX:SemeruConcGCThreads=${ConcThread}
```

More details of *Semeru*'s installation and deployment can be found in *Semeru*'s code repository.

References

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

[4] Libsvm data: Classification. https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets, 2012.

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

[6] P. Agrawal, D. Kifer, and C. Olston. Scheduling shared scans of large data files. *Proceedings of VLDB Endow.*, 1(1):958–969, 2008.

[7] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, S. Novakovic, A. Ramanathan, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote regions: A simple abstraction for remote memory. In *USENIX ATC*, pages 775–787, 2018.

[8] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote memory in the age of fast networks. In *SoCC*, pages 121–127, 2017.

[9] M. K. Aguilera, K. Keeton, S. Novakovic, and S. Singhal. Designing far memory data structures: Think outside the box. In *HotOS*, pages 120–126, 2019.

[10] J. Ahn, S. Yoo, O. Mutlu, and K. Choi. Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *ISCA*, pages 336–348, 2015.

[11] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker. Can far memory improve job throughput? In *EuroSys*, 2020.

[12] Amazon. Amazon EC2 root device volume. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/RootDeviceStorage.html#RootDeviceStorageConcepts, 2019.

[13] S. Angel, M. Nanavati, and S. Sen. Disaggregation and the application. In *HotCloud*, 2020.

[14] Hadoop: Open-source implementation of MapReduce. http://hadoop.apache.org.

[15] Apache Flink. http://flink.apache.org/.

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

[17] K. Asanović, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[18] J. Auerbach, D. F. Bacon, P. Cheng, D. Grove, B. Biron, C. Gracie, B. McCloskey, A. Micic, and R. Sciampacone. Tax-and-spend: Democratic scheduling for real-time garbage collection. In *EMSOFT*, pages 245–254, 2008.

[19] D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *POPL*, pages 285–298, 2003.

[20] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, 2017.

[21] L. A. Barroso. Warehouse-scale computing: Entering the teenage decade. In *ISCA*, 2011.

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

[23] M. N. Bojnordi and E. Ipek. A programmable memory controller for the DDRx interfacing standards. *ACM Trans. Comput. Syst.*, 31(4):11:1–11:31, 2013.

[24] V. R. Borkar, M. J. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, pages 1151–1162, 2011.

[25] Y. Bu, V. Borkar, G. Xu, and M. J. Carey. A bloat-aware design for big data applications. In *ISMM*, pages 119–130, 2013.

[26] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient iterative data processing on large clusters. *PVLDB*, 3(1):285–296, 2010.

[27] T. Cao, S. M. Blackburn, T. Gao, and K. S. McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. In *ISCA*, pages 225–236, 2012.

[28] W. Cao, Z. Liu, P. Wang, S. Chen, C. Zhu, S. Zheng, Y. Wang, and G. Ma. PolarFS: An ultra-low latency and failure resilient distributed file system for shared storage cloud database. *Proc. VLDB Endow.*, 11(12):1849–1862, 2018.

[29] A. Carbonari and I. Beschasnikh. Tolerating faults in disaggregated datacenters. In *HotNets-XVI*, pages 164–170, 2017.

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

[31] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, 2008.

[32] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *PLDI*, pages 363–375, 2010.

[33] I.-H. Chung, B. Abali, and P. Crumley. Towards a composable computer system. In *HPC Asia*, pages 137–147, 2018.

[34] C. Click, G. Tene, and M. Wolf. The pauseless gc algorithm. In *VEE*, pages 46–56, 2005.

[35] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *OSDI*, 1994.

[36] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *ISMM*, pages 37–48, 2004.

[37] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast remote memory. In *NSDI*, pages 401–414, 2014.

[38] Facebook. Introducing Lightning: A flexible NVMe JBOF. https://code.fb.com/data-center-engineering/introducing-lightning-a-flexible-nvme-jbof, 2019.

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

[40] L. Fang, K. Nguyen, G. Xu, B. Demsky, and S. Lu. Interruptible tasks: Treating memory pressure as interrupts for highly scalable data-parallel programs. In *SOSP*, pages 394–409, 2015.

[41] M. J. Feeley, W. E. Morgan, E. P. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *SOSP*, pages 201–212, 1995.

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

[43] C. H. Flood, R. Kennke, A. Dinn, A. Haley, and R. Westrelin. Shenandoah: An open-source concurrent compacting garbage collector for openjdk. In *PPPJ*, pages 13:1–13:9, 2016.

[44] M. D. Flouris and E. P. Markatos. The network ramdisk: Using remote memory on heterogeneous nows. *Cluster Computing*, 2(4), Dec 1999.

[45] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network requirements for resource disaggregation. In *OSDI*, pages 249–264, 2016.

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

[47] L. Gidra, G. Thomas, J. Sopena, M. Shapiro, and N. Nguyen. NumaGiC: A garbage collector for big data on big NUMA machines. In *ASPLOS*, pages 661–673, 2015.

[48] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, pages 599–613, 2014.

[49] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with infiniswap. In *NSDI*, pages 649–667, 2017.

[50] Z. Guo, X. Fan, R. Chen, J. Zhang, H. Zhou, S. McDirmid, C. Liu, W. Lin, J. Zhou, and L. Zhou. Spotting code optimizations in data-parallel pipelines through periscope. In *OSDI*, pages 121–133, 2012.

[51] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker. Network support for resource disaggregation in next-generation datacenters. In *HotNets*, pages 10:1–10:7, 2013.

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

[53] R. L. Hudson, R. Morrison, J. E. B. Moss, and D. S. Munro. Garbage collecting the world: One car at a time. In *OOPSLA*, pages 162–175, 1997.

[54] L. Iftode, K. Li, and K. Petersen. Memory servers for multicomputers. In *Digest of Papers. Compcon Spring*, pages 538–547, Feb 1993.

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

[56] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[57] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011.

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

[59] H. Kermany and E. Petrank. The Compressor: Concurrent, incremental, and parallel compaction. In *PLDI*, pages 354–363, 2006.

[60] A. Klimovic, C. Kozyrakis, E. Thereska, B. John, and S. Kumar. Flash storage disaggregation. In *EuroSys*, pages 29:1–29:15, 2016.

[61] A. Klimovic, H. Litz, and C. Kozyrakis. ReFlex: Remote flash ≈ local flash. In *ASPLOS*, pages 345–359, 2017.

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

[63] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *WWW*, pages 591–600, 2010.

[64] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *ASPLOS*, pages 84–92, 1996.

[65] S. Legtchenko, H. Williams, K. Razavi, A. Donnelly, R. Black, A. Douglas, N. Cheriere, D. Fryer, K. Mast, A. D. Brown, A. Klimovic, A. Slowey, and A. Rowstron. Understanding rack-scale disaggregated storage. In *HotStorage*, 2017.

[66] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, Nov. 1989.

[67] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *ISCA*, pages 267–278, 2009.

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

[69] M. Maas, K. Asanović, and J. Kubiatowicz. A hardware accelerator for tracing garbage collection. In *ISCA*, pages 138–151, 2018.

[70] M. Maas, T. Harris, K. Asanović, and J. Kubiatowicz. Taurus: A holistic language runtime system for coordinating distributed managed-language applications. In *ASPLOS*, pages 457–471, 2016.

[71] H. A. Maruf and M. Chowdhury. Effectively prefetching remote memory with Leap. In *USENIX ATC*, pages 843–857, 2020.

[72] 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.

[73] J. Mickens, E. B. Nightingale, J. Elson, K. Nareddy, D. Gehring, B. Fan, A. Kadav, V. Chidambaram, and O. Khan. Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications. In *NSDI*, pages 257–273, 2014.

[74] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. In *OOPSLA*, pages 245–260, 2007.

[75] D. G. Murray, M. Isard, and Y. Yu. Steno: automatic optimization of declarative queries. In *PLDI*, pages 121–131, 2011.

[76] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *SOSP*, pages 439–455, 2013.

[77] M. Nanavati, J. Wires, and A. Warfield. Decibel: Isolation and sharing in disaggregated rack-scale storage. In *NSDI*, pages 17–33, 2017.

[78] C. Navasca, C. Cai, K. Nguyen, B. Demsky, S. Lu, M. Kim, and G. H. Xu. Gerenuk: Thin computation over big native data using speculative program transformation. In *SOSP*, pages 538–553, 2019.

[79] K. Nguyen, L. Fang, G. Xu, B. Demsky, S. Lu, S. Alamian, and O. Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *OSDI*, pages 349–365, 2016.

[80] K. Nguyen, K. Wang, Y. Bu, L. Fang, J. Hu, and G. Xu. FACADE: A compiler and runtime for (almost) object-bounded big data applications. In *ASPLOS*, pages 675–690, 2015.

[81] C. Olston, B. Reed, A. Silberstein, and U. Srivastava. Automatic optimization of parallel dataflow programs. In *USENIX ATC*, pages 267–273, 2008.

[82] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.

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

[84] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *NSDI*, pages 361–378, 2019.

[85] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *ISCA*, pages 13–24, 2014.

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

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

[88] V. Shrivastav, A. Valadarsky, H. Ballani, P. Costa, K. S. Lee, H. Wang, R. Agarwal, and H. Weatherspoon. Shoal: A network architecture for disaggregated racks. In *NSDI*, pages 255–270, 2019.

[89] D. Sidler, Z. Wang, M. Chiosa, A. Kulkarni, and G. Alonso. StRoM: Smart remote memory. In *EuroSys*, 2020.

[90] G. Tene, B. Iyengar, and M. Wolf. C4: The continuously concurrent compacting collector. In *ISMM*, pages 79–88, 2011.

[91] S.-Y. Tsai and Y. Zhang. LITE kernel RDMA support for datacenter applications. In *SOSP*, pages 306–324, 2017.

[92] Storm: dstributed and fault-tolerant realtime computation. https://github.com/nathanmarz/storm.

[93] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *PSDE*, pages 157–167, 1984.

[94] VMware. Virtual SAN. https://www.vmware.com/products/vsan.html, 2019.

[95] C. Wang, H. Cui, T. Cao, J. Zigman, H. Volos, O. Mutlu, F. Lv, X. Feng, and G. H. Xu. Panthera: Holistic memory management for big data processing over hybrid memories. In *PLD*, pages 347–362, 2019.

[96] W.-H. Wang, J.-L. Baer, and H. M. Levy. Readings in computer architecture. chapter Organization and Performance of a Two-level Virtual-real Cache Hierarchy, pages 434–442. 2000.

[97] Wen-Hann Wang, J. Baer, and H. M. Levy. Organization and performance of a two-level virtual-real cache hierarchy. In *ISCA*, pages 140–148, 1989.

[98] M. Wu, Z. Zhao, Y. Yang, H. Li, H. Chen, B. Zang, H. Guan, S. Li, C. Lu, and T. Zhang. Platinum: A cpu-efficient concurrent garbage collector for tail-reduction of interactive services. In *USENIX ATC*, 2020.

[99] M. Wu, Z. Ziming, L. Haoyu, L. Heting, C. Haibo, Z. binyu, and G. Haibing. Espresso: Brewing Java for more non-volatility. In *ASPLOS*, pages 70–83, 2018.

[100] G. Xu. Finding reusable data structures. In *OOPSLA*, pages 1017–1034, 2012.

[101] G. Xu, M. Arnold, N. Mitchell, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. In *PLDI*, pages 174–186, 2010.

[102] G. H. Xu, M. Veanes, M. Veanes, M. Musuvathi, T. Mytkowicz, B. Zorn, H. He, and H. Lin. Niijima: Sound and automated computation consolidation for efficient multilingual data-parallel pipelines. In *SOSP*, pages 306–321, 2019.

[103] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, pages 1–14, 2008.

[104] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. HotCloud, page 10, Berkeley, CA, USA, 2010.

[105] Q. Zhang, G. Yu, C. Guo, Y. Dang, N. Swanson, X. Yang, R. Yao, M. Chintalapati, A. Krishnamurthy, and T. Anderson. Deepview: Virtual disk failure diagnosis and pattern detection for Azure. In *NSDI*, pages 519–532, 2018.

[106] J. Zhou, P.-Å. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *ICDE*, pages 1060–1071, 2010.