MemLiner: Lining up Tracing and Application for a Far-Memory-Friendly Runtime

Chenxi Wang, Haoran Ma, Shi Liu, Yifan Qiao, Jonathan Eyolfson, and Christian Navasca, *UCLA;* Shan Lu, *University of Chicago;* Guoqing Harry Xu, *UCLA*

https://www.usenix.org/conference/osdi22/presentation/wang

This paper is included in the Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-28-1

Open access to the Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

MemLiner: Lining up Tracing and Application for a Far-Memory-Friendly Runtime

Chenxi Wang[†♣] Haoran Ma[†♣] Shi Liu[†] Yifan Qiao[†] Jonathan Eyolfson[†] Christian Navasca[†]
Shan Lu[‡] Guoqing Harry Xu[†]
University of California, Los Angeles[†] *University of Chicago*[‡]

Abstract

Far-memory techniques that enable applications to use remote memory are increasingly appealing in modern data centers, supporting applications' large memory footprint and improving machines' resource utilization. Unfortunately, most far-memory techniques focus on OS-level optimizations and are agnostic to managed runtimes and garbage collections (GC) underneath applications written in high-level languages. With different object-access patterns from applications, GC can severely interfere with existing far-memory techniques, breaking remote memory prefetching algorithms and causing severe local-memory misses.

We developed MemLiner, a runtime technique that improves the performance of far-memory systems by "lining up" memory accesses from the application and the GC so that they follow similar memory access paths, thereby (1) reducing the local-memory working set and (2) improving remote-memory prefetching through simplified memory access patterns. We implemented MemLiner in two widely-used GCs in OpenJDK: G1 and Shenandoah. Our evaluation with a range of widely-deployed cloud systems shows MemLiner improves applications' end-to-end performance by up to **2.5**×.

1 Introduction

Datacenters are becoming increasingly *memory constrained* [65, 45, 40] with the ubiquitous deployment of in-memory data analytics and ML systems like Neo4j [52], Cassandra [12], Spark [74] and TensorFlow [5], which hold large amounts of intermediate data in memory for quick processing. To tackle this constraint, far-memory techniques [30, 10, 63, 58, 26] that enable applications to use remote memory are increasingly appealing, backed by advances in hardware and networking techniques [13, 62, 66, 23, 19, 28, 35, 49, 55, 59, 32, 8, 16, 38, 41, 33, 63, 43, 57, 37, 42, 60, 7] that allow remote memory to offer much lower latency and higher bandwidth than local block devices.

♣ Contributed equally.

Most of these far-memory systems [30, 10, 63, 48, 68] build on a cache-and-swap mechanism: the application's host server uses local memory as a *data cache*. Once a page that does not reside in the local memory is accessed, a page fault is triggered and the page is fetched from a remote server into the local memory. Good locality and effective remote-memory prefetching [50, 48] are crucial to the performance of applications running in such far-memory systems.

Unfortunately, the interference from garbage collection (GC) severely degrades the memory-access locality and remote-memory prefetching for applications written in high-level languages (e.g., Java, Go, and Python), which are dominant in datacenter workloads. At run time, application threads access heap objects following their program-execution paths, while GC threads *concurrently* scan the heap, performing graph traversal from a set of "roots" (*i.e.*, objects referenced by stack and global variables) to mark live objects. Object accesses by these two sets of threads are uncoordinated, creating two disjoint working sets, as illustrated by Figure 1(a), and causing severe performance problems.

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

Problem 2: Ineffective Prefetching. Monitoring the execution of a managed program, an OS-level prefetcher such as [48] cannot recognize clear memory-access patterns and has to give up prefetching. The reason is that, even if the appli-

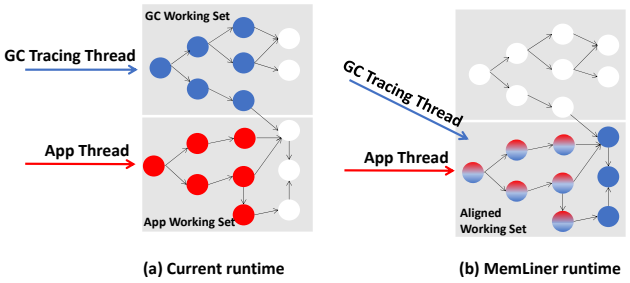


Figure 1: Our main idea: the working sets of GC threads, in blue, and application threads, in red, during a time window (a) without or (b) with the access alignment from MemLiner.

cation's memory accesses follow a simple sequential pattern, the combined accesses from both the application and the GC often appear random from the OS' perspective.

State of the Art. In the past, supporting applications that have large memory footprints (*e.g.*, larger than the main memory size) is not the priority of traditional GC. Although there exists a body of work (such as Platinum [70]) on concurrent GC, such work focuses primarily on improving throughput and reducing latency on memory-abundant servers. However, remote memory is designed to enable applications to use more memory than what their hosts can offer; as a result, developing new GC techniques to support these applications becomes a crucial task.

Recent work Semeru [68] supports running Java programs on disaggregated hardware by disaggregating the traditional JVM into two new ones, with the CPU-JVM executing the program on the CPU server and the memory-JVM performing GC on the memory server. The idea of offloading GC completely to a remote server works for Semeru where *all* the application's memory data is located in a remote server, but does not suit today's datacenters where resources are not entirely disaggregated and applications use remote memory only if their local memory runs out. Furthermore, this offloading approach imposes extra communication overhead for CPU-JVM and memory-JVM to coordinate, and extra computation cost on the remote memory server to run the memory-JVM, which may impose deployment challenges.

Another recent work AIFM [58] proposes a novel runtime to improve the prefetching and swap performance of applications running in remote-memory systems. AIFM targets applications written in native languages (C/C++), and hence cannot easily be applied to solve the GC interference problem in the managed language runtime.

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

The design of MemLiner is based on two key observations.

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

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

Guided by these observations, the key idea behind MemLiner is *working set alignment*. MemLiner carefully reorders the objects traced by the GC threads, so that they follow a similar, although not identical, memory-access path of the concurrent application threads (illustrated by Figure 1(b)). Consequently, their working sets can better overlap with each other; the resource competition can be much alleviated, with much reduced page faults and on-demand swaps; the application's access patterns can be more easily recognized by the underlying prefetcher such as Leap [48]. All of these are achieved in a way that is compatible with existing GC algorithms, without offloading the GC to another machine or re-desgining the prefetcher.

MemLiner must overcome several challenges.

First, *how to align GC threads with application threads*. In a conventional setting, GC traces objects using a graph traversal starting at the root objects. To align GC's accesses with application threads', MemLiner uses a *priority-based* algorithm—MemLiner makes application threads inform the GC of the objects they are accessing; these objects, which must be live and reachable in the object graph at that moment, are then immediately traced and marked by the GC, without any risk of triggering page faults and expensive remote swaps. To enable such communication, MemLiner leverages the *read-write barrier*—a piece of code executed by the runtime at each heap read/write in the application—to inform GC of the objects on the application's access path. Details of the coordination are discussed in §4.1.

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

the application just accessed; the latter is predicted based on object-access history that MemLiner efficiently encodes inside the per-object pointer. Details can be found in §4.2.

Results. We have integrated MemLiner into two widely used GCs (G1 and Shenandoah) in OpenJDK 12. A thorough evaluation with Spark, Cassandra, Neo4J, QuickCached and DayTrade demonstrates that MemLiner improves the end-to-end execution time by an overall of **1.48**× and **1.51**× under the 25% and 13% local memory configurations for the G1 GC, and **2.16**× and **1.80**× for the Shenandoah GC (which runs concurrent GC threads more frequently than G1). Furthermore, MemLiner improves Leap's prefetching coverage and accuracy by **1.5**× and **1.7**×, respectively. Compared to Semeru [68], MemLiner achieves a comparable performance without offloading any computation on remote servers.

Key Takeway. Although there are several directions of work on remote memory (*e.g.*, clean-slate approaches such as AIFM [58] and Kona [17], swap optimizations such as InfiniSwap [30] and FastSwap [10], as well as distributed runtimes such as Semeru [68]), MemLiner takes an *easy-to-adopt, non-intrusive* approach that enables performance improvements for a wide variety of new and legacy applications. MemLiner is orthogonal to (and complements) these existing techniques—aligning the memory accesses between application and GC threads reduces thread-level interference and the application's local-memory working set regardless of the underlying remote-access mechanisms and optimizations.

2 Background

GC. A major benefit of high-level languages over native languages is their support for automated memory management—developers are released from the burden of deallocating objects, leading to improved reliability and security. Automated memory management is enabled by garbage collection (GC), which runs when the heap has little free space. The key idea of GC is simple [36]: perform a reachability analysis to identify a transitive closure of live objects and reclaim objects outside the closure. Consequently, a modern GC algorithm has two main components: (1) *tracing* the heap graph to compute that closure and identify live objects, and (2) *reclamation* of dead objects, while evacuating live objects to contiguous space and updating pointers.

Concurrent Tracing. To ensure the correctness of pointer updating, a conservative way of running GC is to pause all application threads (*i.e.*, a stop-the-world phase) for full-heap tracing and reclamation, which incurs significant delays [53, 47]. To address this performance limitation, starting from the G1 GC [22], which is the default GC in Oracle's JVM, all modern garbage collectors, including Shenandoah [25] from Red Hat and ZGC [2] from Oracle, run the tracing phase concurrently with application threads to (1) leverage the many available cores and (2) minimize GC pauses. For example, in G1, the number of tracing threads is configured, by default, to be 1/4 of the number of cores. Concurrent tracing often uses a snapshot-at-the-beginning (SATB) algorithm [73]—tracing traverses the heap graph from a logical snapshot of the heap; it will not miss any live object as long as object allocation and pointer updates made by the application since the snapshot are recorded and considered conservatively. G1 runs stop-the-world phases to reclaim memory by evacuating live objects into new regions while Shenandoah and ZGC run evacuation also concurrently to minimize the pause time.

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

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

3 Motivation

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

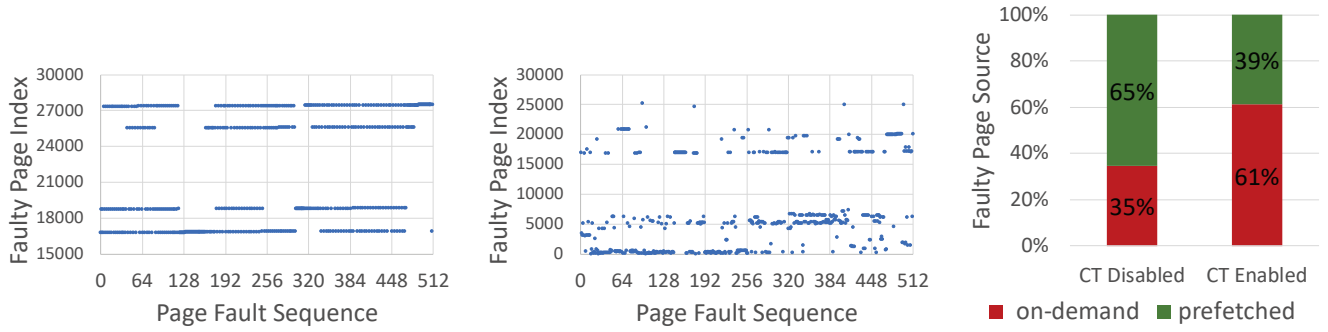
Setup. We ran Spark Logistic Regression (LR) with the Wikipedia dataset on OpenJDK 12 and its default G1 GC. We used two machines, each with 2 Xeon(R) CPU E5-2640 v3 processors, 128GB memory, 1024GB SSD, and CentOS 7.5, connected by RDMA over a 40Gbps InfiniBand network. One machine runs Spark, using local memory and remote memory on the other machine. We configured the first machine to have just enough memory to host 25% of Spark's working set. We name the first server providing compute resource as *host server* and the second server providing remote memory as *remote server*.

We compare the execution of Spark LR in two modes:

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

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

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



(a) Page fault trace of Spark LR. (b) Page fault trace of Spark LR with CT. (c) Prefetching effectiveness.

Figure 2: Prefetching effectiveness for Spark LR executed atop OpenJDK 12 (with its default G1 GC): (a) trace of faulty page index for application threads only; (b) trace of faulty page index when concurrent tracing (CT) is enabled; (c) disabling CT significantly improves the effectiveness of Linux' default swap prefetcher.

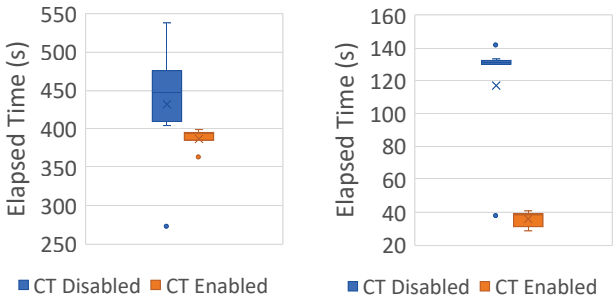
How much interference from concurrent tracing? To have an intuitive look at how well prefetching may or may not work, we randomly sampled 512 *consecutive* page faults in the middle of Spark LR's execution under both execution modes. Note that, since we collected page-fault information from inside the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

Figure 2 (a) and (b) illustrate the virtual page index of the faulty addresses (Y-axis) ordered by when each fault occurs, with the sequence number shown in the X-axis. Without concurrent tracing, each of the application threads has a clear streaming access pattern, as shown in Figure 2(a), which should be detected by an advanced prefetcher. This clear pattern is messed up by concurrent tracing, as shown in Figure 2(b), making prefetching much harder.

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

As shown in Figure 2(c), without concurrent tracing, prefetching is effective, addressing 65% of the page faults. Unfortunately, with concurrent tracing, this ratio greatly dropped to only 39%, with the remaining 61% of page faults leading to costly remote-memory accesses. Note that our experiments use Linux's default swap prefetcher. If an advanced prefetcher such as Leap [48] is used, the prefetch-ratio would be even higher without concurrent tracing and hence suffer even more from the interference (see §7).

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



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

Figure 3: Concurrent tracing improves overall performance. (Data is from 10 runs of each program; dots are outliers.)

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

Unfortunately, this strawman solution does not work. First, modern concurrent GCs such as Shenandoah [25] and ZGC [2], which are designed for low-pause and used widely by latency-sensitive cloud applications, rely on concurrent tracing to reclaim memory (also concurrently). Disabling concurrent tracing would destroy the functionality of such collectors. Second, even for GCs such as G1 that could perform tracing in a stop-the-world phase, the end-to-end execution time suffers significantly without concurrent tracing. As shown in Figure 3(a), the execution time increases by **18%** on average in 10 runs. The main reason is that the aggregated stop-the-world GC periods now take **2.7**× longer without concurrent tracing, as shown in Figure 3(b). Without concurrent tracing, each (fast young-generation) GC cannot

reclaim as many dead objects in the same amount of time and has to resort to *slow, full-heap GC* that scans and compacts the whole heap space in a stop-the-world period, which is extremely time consuming. For example, the longest full-heap GC (*i.e.*, a single pause) in Spark LR takes 76.9 seconds, clearly an intolerable delay.

Key Takeway. Memory accesses from application and GC threads exhibit diverse patterns, significantly increasing the application's working set and making prefetching harder. Simply disabling concurrent tracing in GC would not work, as it reduces the number of local-memory misses at a cost of significantly increased GC pause and end-to-end execution time. MemLiner offers a solution that can greatly reduce the number of local-memory misses and increase the effectiveness of existing prefetchers without introducing extra GC-pause time, and hence effectively reduce the end-to-end execution time.

4 MemLiner Design and Implementation

This section presents the design and implementation of MemLiner, particularly how we realize the two key ideas: (1) making GC concurrently trace objects immediately after their access by application threads (§4.1) and (2) making GC trace other live objects through a novel priority-based algorithm (§4.2) to reduce interference.

MemLiner modifies the garbage collector inside the runtime and the swapping system inside the kernel, while requiring *no* changes to applications. In terms of runtime changes, MemLiner is a general mechanism that can be integrated into any modern runtime that performs concurrent tracing. This paper focuses on a design for Oracle's OpenJDK, a commercial JVM that supports a variety of high-level languages such as Java, Scala, Python, Ruby, *etc.* In terms of kernel changes, we build MemLiner atop paging/swap mechanisms that already exist in the OS kernel, with minimal invasion. Any swap optimizations such as InfiniSwap [30] and FastSwap [10] can be readily used to improve the swap performance for a MemLiner-equipped runtime. MemLiner's runtime design is independent of how remote memory is accessed; for example, MemLiner could also run on a clean-slate platform such as Kona [17] that access remote memory based on cache coherence, not page faults, if coherence is provided by hardware.

When a MemLiner-equipped JVM is launched, the maximum heap size M is specified by the user via a command-line option. A small amount of physical memory on the local machine is initially used to back up the heap (which is much smaller than M). The heap stays entirely in local memory until its usage exceeds the size of local memory, in which case, the OS kernel allocates remote memory by registering it as an RDMA buffer. The kernel uses an approximate LRU algorithm to evict pages. MemLiner does not require any software/hardware support on remote servers, providing a practical solution that can be readily used in today's cloud.

4.1 Application and GC Coordination

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

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

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

To minimize the maintenance overhead, we represent each PQ as a non-blocking *ring buffer*. Producers and consumers do not synchronize at all—an application thread keeps writing into the queue even if it is full. As such, the application thread may overwrite entries that have not yet been picked up by GC. Note that this would not cause any correctness issues because those entries only indicate *tracing priority*: overwriting an entry will delay the corresponding object's tracing, but the tracing of these objects will eventually happen in GC's regular graph traversal, which will be discussed in the next sub-section.

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

4.2 MemLiner Tracing Algorithm

4.2.1 Design Overview

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

will delay the closure computing, leading to inefficiencies in memory reclamation.

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

Reachable Object Classification. To better explain our tracing algorithm, we first classify all live objects at any moment of the execution into three categories based on their location and when they are accessed by the application, as illustrated in Figure 4[1]:

*(1) Objects in local memory (*i.e.*, data cache):* These objects have recently been accessed by the application and have not been evicted yet. Clearly, tracing them at this moment (or in the near future) would not generate any page faults or interfere with the application. Many of these object (*i.e.*, the red ones in the figure) are made known to the GC through the PQ discussed in §4.1. However, since the PQ is designed to be a ring buffer, some of these objects (*i.e.*, the striped ones in the figure) may be missed by GC due to being overwritten in the ring buffer. How to trace them sooner rather than later requires extra handling that we will discuss later.

(2) Objects in remote memory and to be used soon: Since these objects (*i.e.*, the wavy nodes in Figure 4) will soon be accessed by the application, they are typically just a few references away from the objects being accessed by the application. Tracing them is also desirable—although they are currently not local, they will soon be needed by the application. If GC triggers page faults when accessing them, the costs of handling these faults and swapping would be *necessary* as they are "prepaid" by GC for the application.

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

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

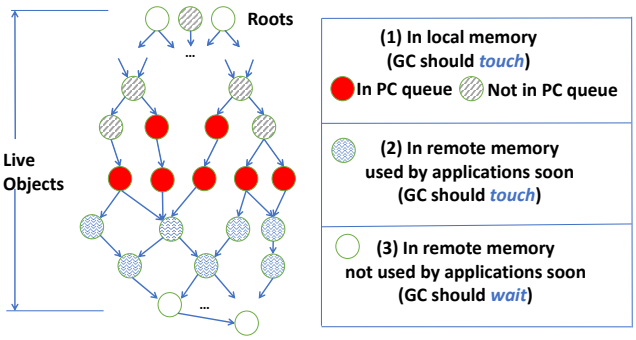


Figure 4: Classification of reachable objects in the heap: red objects are being accessed by the application and shaded objects are what MemLiner intends to trace.

With the red objects in hand, the wavy objects in Category (2) are just a few references away. To mark these objects, we let GC trace *a small number of references forward* from the red objects, which were retrieved from the PQs. As discussed above, tracing such an object will likely trigger swapping, prepaying the cost for the application to access the object soon later. Note that tracing too many references forward will not be useful, as that may bring in objects not used by the application in the near future. In our implementation, we limit the number of hops to 3, which is often large enough to cover objects in the same logical data structure [72].

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

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

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

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

Different from a traditional GC, MemLiner modifies the traversal algorithm to consider whether an object o to be traced is likely in local memory (*i.e.*, whether o is a striped Category (1) object or a Category (3) object)—if o is *estimated* to reside in local memory (*i.e.*, a striped Category (1) object), it is traced right away in GC; if not (*i.e.*, Category (3)), MemLiner postpones processing o in its graph traversal until a later time, optimistically hoping that o will be used by the application before it is encountered again in GC. After postponing a number of times (referred to as MAX_DL below) , GC processes o even if it is still estimated to be remote, so that the closure computation will not be significantly delayed. MemLiner dynamically adjusts the value of

[1]For ease of discussion, here we do not consider cold objects staying in cache due to hot objects on the same page. We will discuss it in Section 4.3.

Figure 5: A 64-bit object pointer in MemLiner.

MAX_DL, in response to the size of available heap space. For example, when the available heap size is in the *red zone* (*i.e.*, <15% available space), MAX_DL will be set to 0, letting GC quickly finish tracing and collect memory. Details of this adaptive algorithm can be found in this section.

4.2.2 Object Location Estimation

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

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

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

Timestamp. In the JVM, virtual addresses of objects are represented as *references*, which are essentially pointers with a strong type. In a 64-bit JVM, the format of an object reference is shown in Figure 5. Recall that our need is to estimate whether an object is in local memory from a reference/pointer of the object (*e.g.*, recorded in a field of another object) during GC's graph traversal. Our idea here is to modify the pointer format by reserving 4 unused bits as a *timestamp* (ts in Figure 5) that indicates *the epoch in which the pointer was last dereferenced*—once the epoch ID reaches 15, the next epoch ID goes back to 0. Dereferencing the pointer accesses the target object (*i.e.*, bringing the object to local memory if it is remote). As such, if the timestamp is close to the current epoch, the object is likely in local memory (*i.e.*, Category (1)) and GC should follow the pointer to trace the object; otherwise, the object may not be local (*i.e.*, Category (3)), and GC should postpone tracing it.

Algorithm 1: Allocation semantics.

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

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

Algorithm 2: Object read and write semantics in application threads.

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

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

Upon the allocation of a new object o, MemLiner sets the timestamp bits in o's pointer to be the current epoch number (with function UPDATEPOINTER in Algorithm 1).

Whenever an object is read/written in an application thread like $b.f = a$ or $a = b.f$ (Algorithm 2), MemLiner updates the timestamp ts in the dereferenced pointer b to be the current epoch ID. Furthermore, if a and $b.f$ are also object references, we write an updated pointer of a into $b.f$, indicating that soon the object referenced by $b.f$ will be accessed through a. Again, this instrumentation is implemented through read-/write barriers.

Note that we use Algorithm 1 and Algorithm 2 to illustrate the high-level logic. Our implementation actually inserts assembly code for efficiency. Changing object pointers in the JVM would not cause problems for actual memory accesses—although each pointer represents a virtual address, the barriers we use mask pointers so that only the last 42 bits are used to access memory.

4.2.3 MemLiner Tracing Algorithm

Algorithm 3 shows GC's tracing logic, which was summarized in §4.2.1. The algorithm takes two queue data structures as input: TQ is a standard tracing queue (already used by the JVM) that contains references yet to be explored in object graph traversal; it is initialized with a set of object references in the stack and global variables (*i.e.*, roots). PQ, as discussed earlier, is the producer-consumer queue that contains references of red objects sent to GC by application threads.

As discussed in §4.2.1, every tracing thread of MemLiner alternates between two modes. In the default mode, tracing loops over the tracing queue TQ, shown in Line 2-13 in Algorithm 3, to perform normal graph traversal. Whenever PQ is not empty (Line 3), the tracing thread interrupts the normal traversal and switches to the other mode to handle the (red) objects in PQ (Line 4); this logic is listed in Algorithm 4 and will be discussed shortly.

Algorithm 3: Main tracing logic in MemLiner's GC.

Input: (1) Producer-consumer queue PQ; (2) tracing queue TQ.
Output: Fully marked live bitmap for all live objects.

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

Algorithm 4: Tracing logic for red and Category-(2) objects.

Input: (1) Producer-consumer queue PQ; (2) regular tracing queue TQ.

```
1  Function TRACEREDANDCATEGORY2(TQ, PQ):
2      while PQ ≠ ∅ do
3          o ← DEQUEUE(PQ)
4          EXPLORE(o, TQ, 0)
```

Input: (1) Object reference o; (2) tracing queue TQ; (3) current exploration depth $depth$.

```
5  Function EXPLORE(o, TQ, depth):
6      MARKLIVEBITMAP(o)
7      foreach Non-null reference-type field f ∈ o do
8          Object reference p ← o.f
9          if depth < MAX_Depth then
10             EXPLORE(p, TQ, depth + 1)
11         else
12             ENQUEUE(TQ, ⟨p, 0⟩)
```

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

The other mode of tracing red objects is triggered when PQ is not empty, as illustrated in Algorithm 4. Similar to the default tracing loop, each iteration of the loop (Line 2) in Algorithm 4 retrieves an object reference from PQ, calling a recursive function EXPLORE to not only mark red objects themselves, but also trace a few references forward to mark objects in Category (2), which may be soon used by the application. We use a recursive function here to control the number of references (*i.e.*, data structure depth) to be explored—once $depth$ exceeds a constant *MAX_Depth* (Line 9, 3 by default), the function does not further explore the object graph, but instead, pushes these unexplored references into the regular tracing queue TQ (Line 12) so that they can be traced later in a normal graph traversal without priority. This is because, as discussed in §4.2.1, following long reference chains can swap in objects that may not be needed by the application in the near future, leading to wasted efforts.

Marking an object live flips its corresponding bit in a global live bitmap (Line 6); as a result, the regular graph traversal (Algorithm 3) would not mark it again if it is encountered there. Once the tracing of the red and Category-(2) objects is done, GC resumes the normal graph traversal in Algorithm 3.

In modern GC with concurrent tracing, each tracing thread works on its own tracing queue TQ. MemLiner modifies each tracing thread to run Algorithm 3 so that the work on TQ is interrupted if there are outstanding red objects in a PQ. Each application thread independently pushes red objects into its thread-local PQ while each tracing thread can consume objects from all PQs. This design makes it possible to enable *work stealing* between threads to balance the number of red and Category-(2) objects processed by these threads. The read/write barrier is already used in existing GC algorithms, such as G1, Shenandoah and ZGC, as well as other far-memory techniques such as AIFM [58]. To further reduce MemLiner's overhead at each read/write barrier, we only need to push the object reference o (64 bit) onto the queue with a very small number of instructions.

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

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

4.3 Discussion

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

For (1), MemLiner uses the timestamp mechanism to reduce the interference between GC and application threads. For example, if the cached pages rarely change (*i.e.*, the application has excellent locality or the local memory size is large enough), the interference is minimal and hence it would not create performance issues if MemLiner does not deviate much from an existing GC. Indeed, our algorithm makes the global epoch change slowly and timestamps in most pointers are the same as the current epoch ID. Algorithm 3 would trace most objects in *TQ* without delays. This is a desired property—when resources are *not* constrained, MemLiner would not incur overhead because GC can trace objects and reclaim memory in a timely fashion.

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

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

5 GC-Specific Optimizations

We have implemented MemLiner in both the JVM's default G1 GC [22] and Red Hat's Shenandoah GC [25], which are two representative GCs widely used in cloud settings. G1 is a generational GC that optimizes for throughput with stop-the-world pauses while Shenandoah is a concurrent GC that minimizes the time of each pause by concurrently tracing and compacting objects. Shenandoah optimizes for latency at the cost of reduced throughput. Our goal is to demonstrate that MemLiner can be easily integrated into both GC algorithms, providing performance benefits for different kinds of (*e.g.*, latency-sensitive or batching) workloads.

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

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

Second, the original G1 only uses the write barrier. Naïvely adding the read barrier into G1 can cause a much higher overhead. We develop the following three optimizations that successfully filter out a significant fraction of object accesses:

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

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

Optimization #3: Our read barrier does not need to update timestamps for objects whose pointer timestamp is the same as the epoch ID. Essentially, we use a check that first compares the pointer timestamp with the epoch ID and updates the timestamp only if they do not have the same value. The larger the local memory percentage is, the less frequently the epoch changes and hence more objects can benefit from this optimization. This explains why when the percentage of local memory increases, MemLiner's overhead does not increase proportionally (as shown in Figure 7).

6 Limitations

MemLiner is designed for managed applications running on a managed runtime and thus not applicable to native applications such as those written in C/C++. Furthermore, MemLiner is designed to optimize throughput (by reducing interference and improving prefetching), *not* latency. However, it does not increase the application latency (*i.e.*, making remote access longer) or the GC pause time. For the Shenandoah GC, its pauses are already very short because operations requiring a pause do not involve many remote accesses and their time is not changed much by MemLiner. For G1, by lining up the tracing and application's memory accesses, MemLiner makes concurrent tracing more efficient, thereby significantly reducing the frequency of triggering full-heap collections. However, it does not reduce the per-collection pause time.

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

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

7 Evaluation

7.1 Experiment Setup

We implemented MemLiner on top of OpenJDK 12 (v 12.0.2) and Linux (v 5.4.0). Our swap system is based upon our re-implementation of FastSwap [10][2], which provides good swap performance. We implemented it on top of G1 and Shenandoah. Implementing MemLiner in other GCs would be straightforward in the future.

Environment. We ran our experiments with two machines, each with two Xeon(R) CPU E5-2640 v3 processors, 128GB memory, one 1TB SSD, and one 40 Gbps Mellanox ConnectX-3 InfiniBand network adapter. They are connected by one Mellanox 100 Gbps InfiniBand switch. One machine runs the JVM process while the other provides remote memory via RDMA. All our experiments used a 32GB heap and 4K pages.

Although our application heap size is relatively small (compared to the size of main memory on our machines), the performance of a remote-memory application depends on how much of its working set can fit into local memory and how many (application and GC) threads are used, *not* on how large local memory is. In particular, MemLiner's key data structure is a per-thread PQ (*i.e.*, TQ is not key to MemLiner as it is GC's original data structure). PQ's size depends on the ratio between the number of applications and the number of tracing threads. For instance, for G1, we follow Oracle's recommendation [56] by setting the number of parallel GC threads to be $5 \times$ (core number)/8, and the number of concurrent tracing threads to be 1/4 of the parallel GC threads. With this ratio and a per-thread PQ of 1024 entries, we rarely saw overwrites in our experiments (with our filtering optimizations stated above). However large the heap is, as long as this ratio remains the same, the size of PQ does not need to change; so does the work done by MemLiner.

[2]Its original implementation was incompatible with OpenJDK12.

Spark [74]	**Dataset**	**Size**
MLlib KMeans (**SKM**)	Wikipedia France [4]	1.1GB
Spark Linear Regression (**SLR**)	Wikipedia English [4]	3GB
Spark Transitive Closure (**STC**)	Synthetic graph	1.5M edges 384K vertices
Cassandra [12]	**Workload**	**Operation**
Update Intensive (**CUI**)	Update 50% Insert 50%	10M ops
Read Intensive (**CRI**)	Read 50% Insert 50%	10M ops
Insert Intensive (**CII**)	Insert 50% Update 25% Read 25%	10M ops
Neo4j [52]	**Dataset**	**Size**
PageRank (**NPR**)	Wikipedia Turkish [4]	14M edges 544K vertices
Triangle Counting (**NTR**)	Wikipedia Turkish [4]	14M edges 544K vertices
Degree Centrality (**NDC**)	Dogster Friends [4]	8.5M edges 451K vertices
QuickCached [3]	**Workload**	**Operation**
Write Dominant (**QWD**)	Insert 60% Read 40%	9M ops
Read Dominant (**QRD**)	Insert 20% Read 80%	9M ops
DayTrader [34]	**Workload**	**Size**
Tradesoap (**DTS**)	Synthetic set of stocks	12288 users 8192 sessions

Table 1: Applications and datesets used for G1.

Applications. To evaluate MemLiner, we used a range of cloud applications including Apache Spark [74] (3.0.0), the de-facto data analytics system, Apache Cassandra [12] (3.11), a widely used distributed database, Neo4j [52] (4.3.2), a graph database, QuickCached [3], a Java implementation of Memcached, as well as DayTrader [34], IBM's open-source application emulating an online stock trading system. These applications cover a wide spectrum of text and graph analytics, web services, machine learning tasks, and database query tasks. For each application, their workloads and datasets are reported in Table 1.

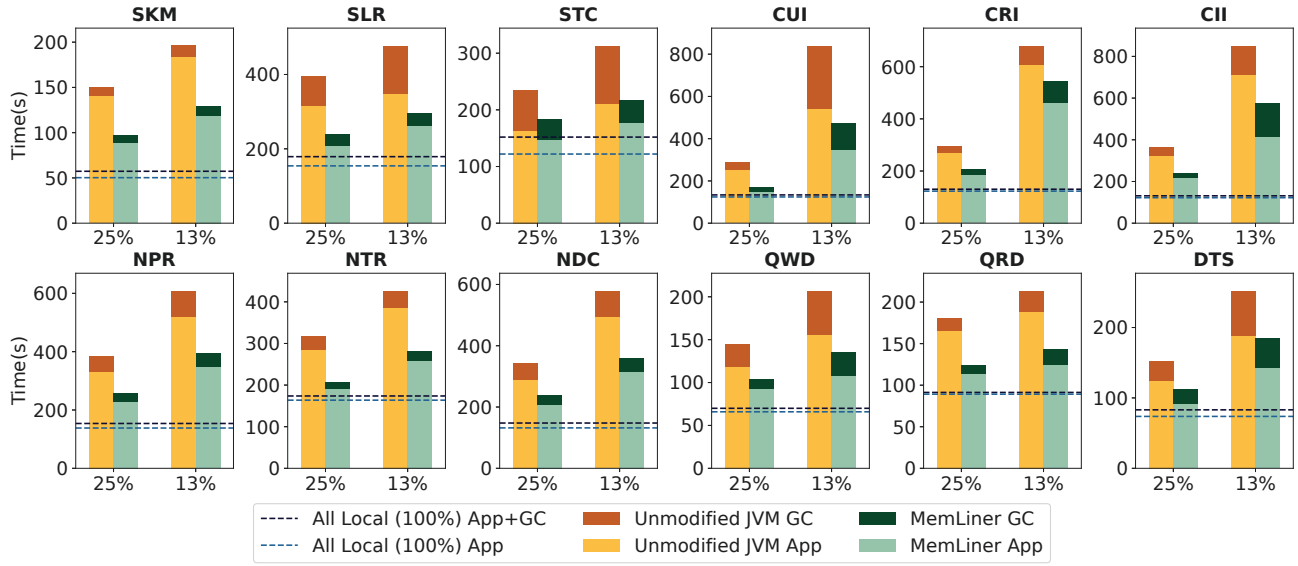


Figure 6: Performance comparisons between G1 GC (yellow bars) and MemLiner (green bars) under two local memory ratios: 25% and 13%; each bar is split into application (bottom with light colors) and GC (top with dark colors) time in seconds. The two dashed lines show application time and total time with unmodified JVM and 100% local memory (no swaps).

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

- *Mostly sequential access patterns*: Spark applications operate over RDDs. An RDD is an object array or serialized primitive array. Each application thread exhibits clear memory access patterns, *e.g.*, streaming or stride.
- *Random access patterns*: QuickCached (a key-value store) and DayTrader (stock trading simulation) exhibit quite random memory access patterns.
- *Mixed access patterns*: Take Cassandra as an example. Each read/update operation goes through several micro-operations. Different micro-operations have different memory access patterns, *i.e.*, the MemTable loading exhibits a good streaming memory access pattern and some other calculations access memory randomly. Both Cassandra and Neo4j belong to this category.

Our experiments considered two local memory ratios: 25%, and 13% of the total Java heap size (32GB), which are consistent with local memory ratios used in prior work [58, 68]. We enforced these ratios with `cgroup`.

7.2 Performance with G1 GC

Overall. Figure 6 compares the performance of the baseline (the default G1 GC) and MemLiner under two different local memory ratios: 25%, and 13%. As shown, MemLiner offers better performance over the baseline JVM for all workloads, **1.48**× speedup on average under 25% local memory and **1.51**× speedup on average under 13% local memory. A summary of these performance improvements (for the application, GC, and end-to-end performance) is reported in Table 2.

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

Table 2: Speedups provided by MemLiner for G1 and Shenandoah. (speedup: the average time under each configuration using the unmodified JVM divided by that using MemLiner)

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

Compared with running the whole application in local memory with no swapping (illustrated by dashed lines in Figure 6), the unmodified JVM incurs 2.17× and 3.73× slowdowns under the 25% and 13% local memory configurations, respectively. MemLiner brings them down to 1.47× and 2.48×.

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

tracing to quickly compute the closure by following the applications' accesses and reducing full-heap GCs.

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

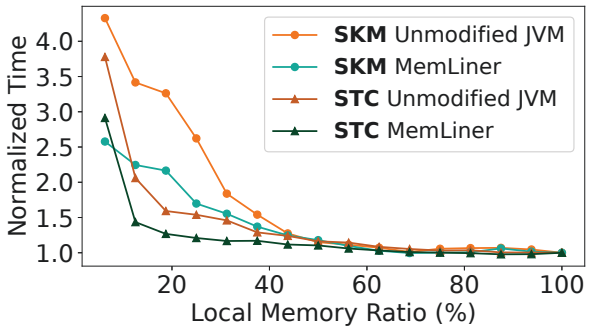


Figure 7: Performance comparisons for **SKM** and **STC** between the unmodified JVM and MemLiner under different local memory configurations.

Different Local Memory Configurations. We ran **SKM** and **STC** with various local-memory ratio configurations and report the performance in Figure 7. As shown, the lower the ratio, the higher the benefit MemLiner provides. For both applications, the turning point is around 50%—MemLiner and the baseline have about the same performance when the local memory ratio reaches 50% or above.

7.3 Performance with Shenandoah GC

To demonstrate the generality of MemLiner, we implemented MemLiner in a second garbage collector: Shenandoah[25], a widely-used highly-concurrent low-pause GC developed by Red Hat. It performs not only concurrent tracing but also concurrent object evaluation to minimize pauses.

Shenandoah provides great latency benefits under sufficient local memory. However, it has extremely poor performance with remote memory involved. For example, the slowdowns under 25% memory for our Spark and Neo4j applications are constantly above 10× and 4×, respectively. Compared to Neo4j, Spark applications usually have much larger working sets, leading to more remote accesses. Such a large overhead highlights the problem of running many concurrent GC threads that do not align with the application's memory access. In particular, Shenandoah is *not* a generational GC (while G1 is). In G1, when the young generation, which contains short-lived objects, is full, the JVM suspends application threads

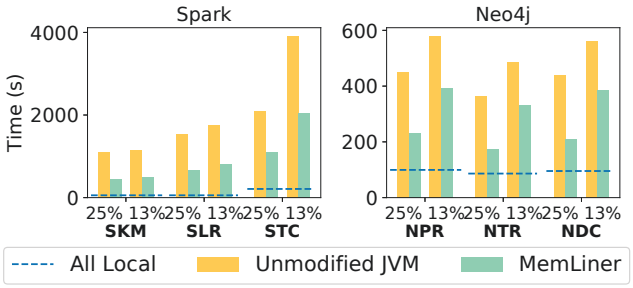


Figure 8: Performance comparison with Shenandoah GC [25].

Spark Programs	Dataset	Size
MLlib KMeans (**SKM**)	Wikipedia Polish [4]	1GB
Spark Linear Regression (**SLR**)	Wikipedia Polish [4]	1GB
Spark Transitive Closure (**STC**)	Synthetic Graph	1.5M edges 384K vertices
Neo4J Programs	**Dataset**	**Size**
PageRank (**NPR**)	Wikipedia Slovak [4]	7.6M edges 291K vertices
Triangle Counting (**NTR**)	Wikipedia Slovak [4]	7.6M edges 291K vertices
Degree Centrality (**NDC**)	Wikipedia min-nan [4]	4.4M edges 429K vertices

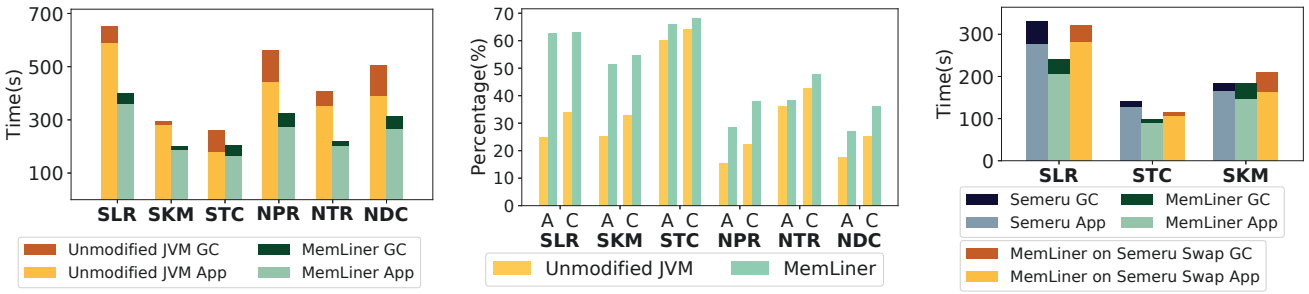
Table 3: Benchmarks and datasets for Shenandoah.

and evacuates objects in the young generation. This leads to excellent data locality after evacuation. However, under Shenandoah GC, the JVM runs concurrent tracing much more frequently to scan the full heap to identify and collect garbage. Those tracing threads exhibit particularly poor locality. To evaluate Shenandoah, we had to use smaller datasets (Table 3) for a tolerable running time.

As illustrated in Figure 8 and summarized in Table 2, MemLiner achieves an overall **2.16**× and **1.80**× speedup compared to the unmodified JVM under 25% and 13% local memory, respectively. MemLiner reduces an average of 82% on-demand swap-ins and 56% of total swap-ins under 25% local memory, while it reduces 79% of on-demand swap-ins and 22% of total swap-ins under 13% local memory. As shown in Table 2, MemLiner provides tremendous improvements for Shenandoah's GC performance, because the unmodified JVM frequently triggers *full-heap stop-the-world GC*.

7.4 Comparisons with Other Systems

Leap [48] is an advanced OS-level prefetcher. It uses a majority-vote algorithm to determine how to do prefetches. In cases where no clear access patterns are seen, Leap aggressively prefetches consecutive pages. Although this strategy may improve performance for native applications whose memory accesses often fall into large arrays, it often hurts managed applications such as Spark, as GC's pointer-chasing behavior often makes prefetched consecutive pages useless.



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

Figure 9: Performance comparisons with Leap and Semeru; Semeru crashed on **NPR**, **NTR**, and **NDC** (*i.e.*, Neo4j applications).

Our hypothesis is that even aggressive prefetchers like Leap cannot handle the interference of GC, and that by aligning the memory accesses of GC with application threads, MemLiner can improve application performance under Leap just like under less aggressive prefetchers. To test our hypothesis, we compared MemLiner with the unmodified JVM (default G1 GC) both using Leap as the prefetcher. This experiment was conducted on three Spark applications: **SLR**, **SKM**, **STC**, and three Neo4j applications: **NPR**, **NTR**, **NDC**, under 25% local memory.

As shown in Figure 9(a), compared with the unmodified JVM on Leap, MemLiner improves the overall performance by an average of 1.6× and reduces 58% of on-demand swap-ins, as well as 53% of total swap-ins on average. To understand whether MemLiner improves Leap's prefetching effectiveness, we additionally measured Leap's prefetching *accuracy* (i.e., the percentage of page faults hitting on the swap cache among prefetched pages) and *coverage* (i.e., the percentage of swap cache hits among all page faults) with and without MemLiner. As shown in Figure 9(b), MemLiner helps Leap deliver higher accuracy and coverage. We still observed that MemLiner is not as useful for **STC** and **NTR** as it is for the two applications. This is because the number of live objects in **STC** during concurrent tracing is relatively small, leading to shorter tracing time and better access patterns. For **NTR**, its application threads exhibit random memory accesses themselves. Hence, Leap cannot detect clear patterns even if MemLiner has already eliminated much of the interference.

Semeru [68] is a memory-disaggregated runtime, where the entire Java heap is backed by physical memory on memory servers and the CPU server's local memory is used as an inclusive cache. Semeru completely redesigned the JVM so that all the garbage collection is offloaded from the CPU server to the memory servers, through special lightweight JVMs running there. Applications execute on the CPU server with absolutely no GC interference, at the cost of extra computation on memory servers (i.e., two extra cores for each memory server to run the offloaded lightweight JVM).

Here, to evaluate whether MemLiner can achieve similar performance as Semeru, without Semeru's intrusive changes to JVM and Semeru's extra computation load on memory servers, we ran the same three Spark applications under 25% local memory on top of (1) Semeru, (2) MemLiner on Semeru's swap system (*i.e.*, a modified version of NVMe-over-fabrics [1]), and (3) MemLiner on FastSwap [10], which is the default swap system MemLiner builds on. We ran Semeru with one CPU server and two memory servers—the Java heap is partitioned between the memory servers.

As shown in Figure 9(c) , MemLiner's performance is comparable with Semeru when using Semeru's swap system, and is much better than Semeru when using MemLiner's default swap system. The reason is that, even though Semeru completely eliminates GC tracing threads from the local machine, it has to perform a great deal of coordination between servers to handle cross-server references, incurring communication overheads. We would have also liked to run Semeru directly over FastSwap, but this was not feasible due to Semeru's runtime-kernel co-design that prevents Semeru from easily adapting to different swap systems.

We could not directly compare Memliner with AIFM [58] as AIFM targets native languages (C/C++) applications and requires rewriting programs. However, the major idea behind AIFM—swapping at the object granularity—is orthogonal to MemLiner. MemLiner can also benefit from a redesigned swap system that performs object-level swapping.

7.5 More Detailed Results

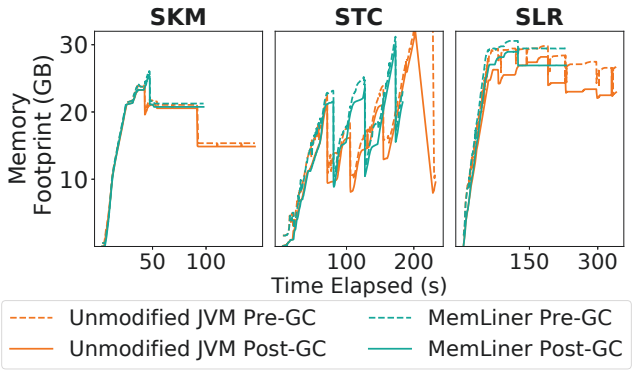


Figure 10: Memory footprints for **SKM**, **STC**, and **SLR**, between unmodified JVM and MemLiner under 25% rate.

Memory Reclamation Impact. Since MemLiner postpones tracing objects estimated to be remote, it may delay memory reclamation. To understand the impact of such a delay, we collected post-GC memory footprints for **STC**, **SKM**, and **SLR** executed atop the unmodified JVM and MemLiner under 25% local memory configuration. Figure 10 reports, for each program, both its pre-GC and post-GC memory footprints. As shown, for all three workloads, MemLiner incurs insignificant delays in memory reclamation and only a slight increase in the peak memory consumption. This is because tracing of each remote object can only be postponed a few times (*i.e.*, *MAX_DL*); when the available heap runs low, *MAX_DL* becomes 0 and we do not postpone GC at all.

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

8 Related Work

Far Memory. Due to rapid technological advances in network controllers, it has become practical to reorganize resources into disaggregated clusters [32, 18, 27, 15]. 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" [14, 44, 43, 67, 20, 38, 7, 11]. A body of techniques [10, 30, 6, 58, 68, 61, 63, 68, 31, 69] have been developed to enable applications to use remote memory and efficiently access remote data.

Among these techniques, a mainstream approach [10, 30, 6] is to provide *transparent* remote memory access with *swap mechanisms* where the running application is not aware of remote memory, which is mapped into the application host server as a *swap partition*. The host server reserves a certain amount of local memory as a software-managed data cache. Once the program accesses a page that does not reside in the data cache, it triggers a page fault, and the swap system fetches the page from a remote memory server via RDMA.

A traditional swap system was designed for *slow and rare* accesses to disks, *not for fast and frequent* accesses to remote memory via RDMA. Having realized this speed discrepancy, existing techniques have performed a variety of optimizations, *e.g.*, removing redundant block layers [30], leveraging multi-queues [10], or performing per-application prefetching [48], all to maximize the paging/swap efficiency. Despite these commendable efforts, these techniques need to pay a "transparency tax"—since all remote accesses go through the OS kernel, which incurs a non-trivial overhead. To mitigate such a software-introduced overhead, work such as AIFM [58] provides primitives for developers to perform efficient remote access in the user space. AIFM outperforms swap-based techniques by bypassing the kernel data plane. However, to use AIFM, applications have to be rewritten (with new primitives), which can significantly hinder its practical use.

Modern Garbage Collectors. Modern GCs, including Oracle's Garbage-First (G1) GC [22], Red Hat's Shenandoah GC [25], Azul's pauseless GC [21], and C4 [64], all use concurrent tracing. Some also perform concurrent memory compaction [39, 2]. As big data systems gain popularity, there is a line of work that develops systems for applications running on the cloud [24, 53, 54, 51, 67], on NUMA machines [29], as well as using non-volatile memory [67, 71, 9]. Yak [53] is a region-based big-data-friendly GC. Taurus [47] coordinates GC efforts among workers in a distributed system. Facade [54] uses region-based memory management to reduce GC costs for Big Data applications. Gerenuk [51] develops a compiler analysis and runtime system that enable native representation of data for managed analytics systems such as Spark and Hadoop. Espresso [71] and Panthera [67] are designed for systems with non-volatile memory. Platinum [70] aims to reduce tail latency for interactive applications. NUMAGiC [29] is a GC that provides efficiency by considering NUMA features.

Semeru [68] and Mako [46] are both GCs developed for memory disaggregation. While they both achieve superior performance via compute offloading (*e.g.*, running concurrent tracing and evacuation on memory servers), offloading introduces numerous challenges in resource utilization and cluster scheduling. AIFM [58] performs GC-like memory compaction to eliminate dead objects to reduce read/write amplification. This approach is orthogonal to MemLiner, which leverages tracing for prefetching.

9 Conclusion

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

Acknowledgments

We thank the anonymous reviewers for their valuable and thorough comments. We are grateful to our shepherd Aurojit Panda for his feedback. This work is supported by NSF grants CNS-1703598, CNS-1763172, CNS-1764039, CNS-1907352, CNS-1956180, CNS-2007737, CNS-2006437, CNS-2128653, CNS-2106838, CCF-2119184, ONR grant N00014-18-1-2037, and research grants from Facebook, Microsoft, and Cisco.

A Artifact Appendix

A.1 Artifact Summary

MemLiner is a managed runtime built for a memory-disaggregated cluster where each managed application runs on one server and uses both local memory and remote memory located on another server. When launched on MemLiner, the process fetches data from the remote server via the paging system. MemLiner reduces the local-memory working set and improves the remote-memory prefetching by lining up the memory accesses from application and GC threads. MemLiner is transparent to applications and can be integrated in any existing GC algorithms, such as G1 and Shenandoah.

A.2 Artifact Check-list

- **Hardware: Intel servers with InfiniBand**
- **Run-time environment: OpenJDK 12.02, Linux-5.4, Ubuntu 18.04 with MLNX-OFED 4.9-2.2.4.0**
- **Public link: https://github.com/uclasystem/MemLiner**
- **Code licenses: The GNU General Public License (GPL)**

A.3 Description

A.3.1 MemLiner's Codebase

MemLiner contains the following three components:

- the Linux kernel, which includes a modified swap system,
- the Java Virtual Machine (JVM) with MemLiner,
- necessary shell scripts and configuration files.

A.3.2 Deploying MemLiner

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

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

When deploying MemLiner, install the components in the following order: (1) install the kernel and the RDMA module on all participating servers; (2) install the JVM with MemLiner on the server that runs the process; (3) connect the participating 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 MemLiner/Kernel
sudo ./build_kernel.sh build
sudo ./build_kernel.sh install
```

- Install the MLNX OFED driver:

  MemLiner has only been tested on Ubuntu 18.04 with MLNX-OFED-4.9-2.2.4.0. The driver should be installed all participating servers.

```
# @all participating servers
# Remove the incompatible libraries
sudo apt remove ibverbs-providers:amd64
librdmacm1:amd64 librdmacm-dev:amd64
libibverbs-dev:amd64 libopensm5a
libosmvendor4 libosmcomp3 -y

# Download and install the MLNX OFED driver
curl https://content.mellanox.com/ofed/
MLNX_OFED-4.9-2.2.4.0/MLNX_OFED_LINUX
-4.9-2.2.4.0-ubuntu18.04-x86_64.tgz
--output MLNX_OFED.tgz
tar -xzf MLNX_OFED.tgz
sudo MLNX_OFED/mlnxofedinstall
--add-kernel-support

# Enable the openidb and opensmd services
sudo systemctl enable openibd
sudo systemctl start openibd
sudo systemctl enable opensmd
sudo systemctl start opensmd
```

- Configure and install the MemLiner RDMA module:

```
# Assign the IP of a memory server into:
# @CPU server
# MemLiner/rswap/client/rswap_rdma.c
char ip[] = "10.0.0.4"; # IP of memory server
# @memory server
# MemLiner/rswap/server/rswap_server.cpp
const char *ip_str = "10.0.0.4";

# Build the MemLiner RDMA module
# @CPU server
cd MemLiner/rswap/client
make clean && make
# @memory server
cd MemLiner/rswap/server
make clean && make
```

Install the MemLiner (JVM). We next discuss the steps to build and install the MemLiner JVM on the CPU server.

- Download Oracle JDK 12 to build the MemLiner JVM:

```
# @CPU server
# Assume jdk 12.02 is under path:
# ${HOME}/jdk-12.0.2
cd MemLiner/JDK
./configure -with-boot-jdk=${HOME}/jdk-
12.0.2 --with-debug-level=release
make JOBS=32

# Run the applications with the built JVM.
# The built JVM (MemLiner) is under:
MemLiner/JDK/build/
linux-x86_64-server-release/jdk
```

A.3.3 Running Applications

To run applications, we first need to connect the CPU and memory servers. Next, we mount the remote memory pool 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 remote memory via RDMA.

- Launch memory servers:

```
# @memory server
cd MemLiner/rswap/server
./rswap-server
```

- Connect the CPU server with memory servers:

```
# @CPU server
cd MemLiner/rswap/client
./manage_rswap_client.sh install
```

- Set a cache size limit for an application:

```
# For example, create a cgroup with a 9GB
memory limit.
# @CPU server
# Create the cgroup with the name memctl
# $USER is the username of the account
sudo cgcreate -t $USER -a $USER -g
memory:/memctl

# Set the memory limit to 9GB
echo 9g > /sys/fs/cgroup/memory/
memctl/memory.limit_in_bytes
```

- Add a Spark executor into `cgroup`:

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

- Launch the Spark cluster:

  Certain JVM options need to be added to run the MemLiner. We use the Spark as an example here. Please refer to the MemLiner's code repository for more details about how to run other applications.

```
# @CPU server
# Replace the Spark default configuration
cd ${spark-home-dir}/conf
cp MemLiner/config-files/spark-confs/
spark-defaults-memliner.conf
spark-defaults.conf

# Launch the Spark master and worker services
${spark-home-dir}/sbin/start-all.sh
```

- Run Spark applications:

  Specify the Spark application name and local memory ratio, *e.g.*, 25% or 13%, and then execute the applications:

```
# @CPU server
# Para#1 application: lr, km, tc
# Para#2 mem_local_ratio: 25, 13
MemLiner/app-scripts/memliner.sh
${application} ${mem_local_ratio}
```

  More details of MemLiner's installation and deployment can be found in MemLiner'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] QuickCached. https://github.com/QuickServerLab/QuickCached, 2017.

[4] Konect network datasets. http://konect.cc/networks/, 2021.

[5] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *OSDI*, pages 265–283, 2016.

[6] 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.

[7] 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.

[8] 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.

[9] S. Akram, J. B. Sartor, S. M. Blackburn, K. S. McKinley, and L. Eeckhout. Write-rationing garbage collection for hybrid memories. In *PLDI*, pages 62–77, 2018.

[10] 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.

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

[12] Apache. Apache cassandra. https://cassandra.apache.org, 2021.

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

[14] 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.

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

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

[17] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli. Rethinking software runtimes for disaggregated memory. In *ASPLOS*, pages 79–92, 2021.

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

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

[20] L. Chen, J. Zhao, C. Wang, T. Cao, J. Zigman, H. Volos, O. Mutlu, F. Lv, X. Feng, G. H. Xu, and H. Cui. Unified holistic memory management supporting multiple big data processing frameworks over hybrid memories. *ACM Trans. Comput. Syst.*, 2022.

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

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

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

[24] 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.

[25] 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.

[26] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay. Caladan: Mitigating interference at microsecond timescales. In *OSDI*, pages 281–297, 2020.

[27] 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.

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

[29] 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.

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

[31] Z. Guo, Y. Shan, X. Luo, Y. Huang, and Y. Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *ASPLOS*, pages 417–433, 2022.

[32] 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.

[33] Hewlett-Packard. The machine: A new kind of computer. https://www.hpl.hp.com/research/systems-research/themachine/, 2015.

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

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

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

[37] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *SIGCOMM*, pages 295–306, 2014.

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

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

[40] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid, G. Thelen, K. A. Yurtsever, Y. Zhao, and P. Ranganathan. Software-defined far memory in warehouse-scale computers. In *ASPLOS*, pages 317–330, 2019.

[41] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. In *ISCA*, pages 2–13, 2009.

[42] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *NSDI*, pages 429–444, 2014.

[43] 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.

[44] 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.

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

[46] H. Ma, S. Liu, C. Wang, Y. Qiao, M. D. Bond, S. M. Blackburn, M. Kim, and G. H. Xu. Mako: A low-pause, high-throughput evacuating collector for memory-disaggregated datacenters. In *PLDI*, pages 92–107, 2022.

[47] 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.

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

[49] 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.

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

[51] 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.

[52] Neo4j. Neo4j graph data platform. https://neo4j.com, 2021.

[53] 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.

[54] 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.

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

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

[57] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The RAMCloud storage system. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, Aug. 2015.

[58] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay. AIFM: High-performance, application-integrated far memory. In *OSDI*, pages 315–332, 2020.

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

[60] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson. Willow: A user-programmable SSD. In *OSDI*, pages 67–80, 2014.

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

[62] 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.

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

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

[65] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes. Borg: The next generation. In *EuroSys*, 2020.

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

[67] 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 *PLDI*, pages 347–362, 2019.

[68] C. Wang, H. Ma, S. Liu, Y. Li, Z. Ruan, K. Nguyen, M. D. Bond, R. Netravali, M. Kim, and G. H. Xu. Semeru: A memory-disaggregated managed runtime. In *OSDI*, pages 261–280, 2020.

[69] C. Wang, Y. Qiao, H. Ma, S. Liu, Y. Zhang, W. Chen, R. Netravali, M. Kim, and G. H. Xu. Canvas: Isolated and adaptive swapping for multi-applications on remote memory. https://arxiv.org/abs/2203.09615, 2022.

[70] 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*, pages 159–172, 2020.

[71] 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.

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

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

[74] 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.