DiLOS: Do Not Trade Compatibility for Performance in Memory Disaggregation

Wonsup Yoon
KAIST
wsyoon@kaist.ac.kr

Jisu Ok
KAIST
jisu.ok@kaist.ac.kr

Jinyoung Oh
KAIST
jinyoungoh@kaist.ac.kr

Sue Moon
KAIST
sbmoon@kaist.edu

Youngjin Kwon
KAIST
yjkwon@kaist.ac.kr

Abstract

Memory disaggregation has replaced the landscape of datacenters by physically separating compute and memory nodes, achieving improved utilization. As early efforts, kernel paging-based approaches offer transparent virtual memory abstraction for remote memory with paging schemes but suffer from expensive page fault handling. This paper revisits the paging-based approaches and challenges their performance in paging schemes. We posit that the overhead of the paging-based approaches is not a fundamental limitation. We propose DiLOS, a new library operating system (LibOS) specialized for paging-based memory disaggregation. We have revamped the page fault handler to get away with the swap cache and incorporated known techniques in our prefetcher, page manager, and communication module for performance optimization. Furthermore, we provide APIs to augment the LibOS with application semantics. We present two app-aware guides, app-aware prefetching and bandwidth-reducing memory allocator in DiLOS. Through extensive evaluation of microbenchmarks and applications, we demonstrate that DiLOS outperforms the state-of-the-art kernel paging-based system (Fastswap) up to 2.24× and a recent user-level system (AIFM) 1.54× on a real-world data analytic workload.

CCS Concepts: • **Computer systems organization** → **Cloud computing**.

Keywords: memory disaggregation,disaggregated data center,unikernel

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

EuroSys '23, May 9–12, 2023, Rome, Italy

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-9487-1/23/05…$15.00
https://doi.org/10.1145/3552326.3567488

ACM Reference Format:
Wonsup Yoon, Jisu Ok, Jinyoung Oh, Sue Moon, and Youngjin Kwon. 2023. DiLOS: Do Not Trade Compatibility for Performance in Memory Disaggregation. In *Eighteenth European Conference on Computer Systems (EuroSys '23), May 9–12, 2023, Rome, Italy*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3552326.3567488

1 Introduction

Resource disaggregation is rearchitecting datacenter infrastructures to address resource management challenges, including independent scale-out, memory capacity expansion, transparent remote memory access, domain-specific accelerators, emerging persistent media, and fault domain isolation [11]. It splits computation, memory, storage, and accelerators into individual resource pools and allows flexible resource provisioning beyond the boundary of a single machine. The benefits are a key driver for solving the chronic resource under-utilization [24, 64, 70] and memory wall [39] problems in datacenters. Cloud service providers are building disaggregated datacenters (DDCs) to take full advantage of resource disaggregation [39].

Memory disaggregation aims to split computing and memory in DDCs, placing computing and memory in physically separate nodes. A computing node has a large number of computational units, while a memory node provides a large amount of memory with few or no computational units. Memory disaggregation enables a large memory pool shared across computing nodes and overcomes the memory wall in the cluster of traditional server nodes [2, 23, 49, 64]. Fast networking technologies such as RDMA and user-space networking stacks have reduced the remote access latency [69] and accelerated the trend towards memory disaggregation.

Early efforts to adopt memory disaggregation have mostly built it as a kernel feature [2, 23, 39, 49, 64]. These approaches refurbish the existing kernels to migrate pages between the computing node and the memory node. Computing nodes evict pages to memory nodes under memory pressure, unmap them in their page table, and fetch them back on page faults. This paging-based mechanism changes the address mapping behind the scene; applications use disaggregated memory without any modification to their code.

The compatibility of the paging-based approaches, however, comes with a cost. Frequent kernel-user switching to handle page fault exceptions incurs non-negligible performance overhead. In addition, recent work [60] reports the size gap between a page and actual access units (*e.g.*, an object accessed by an application) introduces IO amplification because paging-based approaches fetch an entire page from a memory node.

To overcome these limitations, other approaches take advantage of user-level semantics from applications or language runtimes. AIFM [60] and Carbink [77] put user-level libraries and APIs in charge of remote memory management to avoid page faults. Semeru [72] and MemLiner [73] improve the performance of JVM-based applications by using runtime semantics. Semeru reduces the number of page faults and network bandwidth by offloading JVM garbage collection to its memory node. MemLiner proposes a far-memory-friendly garbage collector that traces recent objects, aligning memory access patterns of tracing and main thread, and in turn, improves remote memory prefetching. However, they trade compatibility for performance. AIFM and Carbink programming models require annotations for remote objects and mandate custom C++ APIs. Semeru and MemLiner only support JVM-related programming languages.

Here, we look back at the burden of the kernel paging-based model: *Is the performance overhead of the kernel-based approaches the fundamental limitation worth giving up compatibility?* The question has led us to rehash existing attempts to curb the paging overhead and the IO amplification. We claim that the performance overhead from paging is not fundamental. In this paper, we demonstrate how to overcome the performance limitation by *specializing the kernel* for memory disaggregation.

To demonstrate our claim, we choose Library OS (LibOS) as our baseline framework. LibOS has the advantage of the simplified code path between an application, an OS, and hardware. The simple code base allows us to focus on building our design ideas efficiently. Also, the absence of user-kernel crossing overhead [32, 34, 57] opens doors for additional improvements by augmenting a LibOS kernel with application semantics. Moreover, by using a POSIX-compatible LibOS, developers need not modify existing applications for memory disaggregation.

In this paper, we propose a LibOS-based memory disaggregation system, DiLOS, that offers both performance and compatibility. DiLOS is paging-based but eliminates unnecessary overhead in its page fault handler. At its core, DiLOS builds its own data-path for memory disaggregation rather than reusing the slow kernel swap subsystem as in existing kernel paging-based systems [2, 23, 49], significantly improving latency when accessing remote pages in its page fault handler. On the base of the fast page fault handler, DiLOS incorporates known techniques for performance optimization, such as prefetching, background write-back, and fast RDMA communication. DiLOS supports POSIX compatibility; it loads the existing binary without any modifications.

In addition, DiLOS provides APIs to integrate application semantics in the form of *guides*. A guide is a pluggable module implemented in the form of a third-party binary (like shared libraries in Linux) without modifying the main code of an application. DiLOS showcases two examples of guides: an app-aware prefetcher and a bandwidth-reducing memory reclamation. The prefetchers take the guide as a cue for further performance improvement over general-purpose prefetchers. The bandwidth-reducing memory reclamation uses hints from a user-level memory allocator to exclude unused areas in a page when migrating pages between a computing node and a memory node.

We implement DiLOS on a mature, open-source LibOS, OSv [32]. Although we have decided to implement DiLOS on a LibOS system, the core design of DiLOS does not exclude Linux as a development platform. Yet, we take full advantage of the small code base of the LibOS system and present DiLOS as a proof-of-concept implementation. We leave implementation on Linux for future work (§5.1).

For evaluation, we compare DiLOS with AIFM and the kernel paging-based system, Fastswap. When a computing node has 12.5% and 100% local memory of the total working set, DiLOS performs 54% and 83% better than AIFM, respectively, even without modifying the applications. Against Fastswap, DiLOS demonstrates up to 2.2× superior performance in real-world workloads. In addition, DiLOS with an app-aware prefetcher achieves up to 62% higher throughput in Redis than DiLOS with a general-purpose prefetcher proposed in recent work [49].

2 Background

There are basically three approaches to memory disaggregation: either at the kernel-level, at the user-level, or at the hardware-level. The first two are implemented in software using conventional NICs (both RDMA and TCP), and the other one uses specialized hardware. In this work, we mainly focus on kernel-level and user-level. The hardware approach is discussed in §7.

The kernel-level memory disaggregation, such as Infiniswap [23], Leap [49], and Fastswap [2], extends Linux's swap subsystem to use remote memory as swap space. However, Linux's swap mechanism involves complex data structures to manage swap space, incurring significant overhead (§3.1). LegoOS [64] proposes a new kernel design by splitting kernel services over disaggregated resources. While LegoOS provides a new building block for resource disaggregation, it also relies on complex general-purpose kernel code, showing moderate performance compared to user-level systems.

User-level systems, on the other hand, offer a language runtime (Semeru [72] and MemLiner[73]) or a library interface (AIFM [60] and Carbink [77]). Semeru and MemLiner

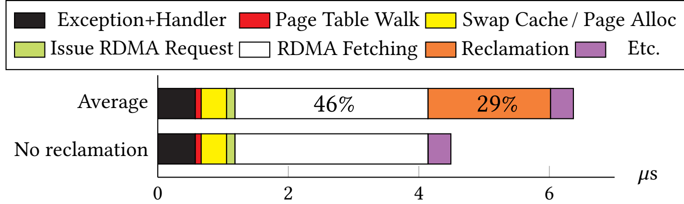


Figure 1. The performance breakdown of Fastswap [2]'s page fault handler. 99% of page faults are handled without reclamation.

modify Java Virtual Machine (JVM) to obtain hints from serspace and gain full knowledge of a Java application's memory layout. With this approach, Semeru offloads memory-intensive parts (garbage collection) to the memory node, and MemLiner reduces the local working set. AIFM offers C++ STL-like interfaces for application developers to integrate remote memory features into their applications. Since AIFM is implemented at the user-level and uses a user-space networking stack, AIFM delivers an efficient data-path avoiding expensive kernel context switching. Moreover, AIFM also has the full knowledge of application semantics, such as the C++ library memory layout, which translates to useful hints for prefetching or offloading memory operations.

3 Motivation

In this section, we review the reported limitations of kernel paging-based approaches [60] and assess the potential for performance improvement in page fault handling.

3.1 Cost of Handling Page Faults

In kernel paging-based systems, the page fault handler lies on the critical path. To understand the cost of page fault handling, we begin with a latency analysis of the state-of-the-art kernel paging-based system, Fastswap [2].

Figure 1 shows the latency breakdown of page fault handling when accessing a remote memory node to fetch a faulted page. Fetching a page will reclaim existing pages in the local DRAM if it has insufficient space. In the analysis, the reclaimed pages are clean, so the kernel does not write back evicted pages to the remote memory node via RDMA, revealing the software cost of reclamation caused by Fastswap. No reclamation in the figure represents a breakdown when eviction does not happen.

In both average and no reclamation cases, the most significant portion of the latency stems from fetching a remote page: 46%. Fetching a remote page via RDMA is unavoidable regardless of the approaches, whether kernel paging-based or user-level. Another inevitable delay is from handling hardware page fault exceptions (hardware exception delay + OS exception handler), which takes up 9% (0.57 μs). The remaining portions come from the executing OS code of the page fault handler. Fastswap performs direct page

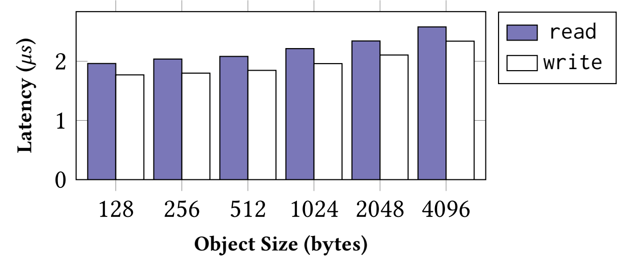


Figure 2. RDMA latency (μs) for a range of object sizes. One-sided operations are used.

	Count	%
Major page fault	655,737	12.5
Minor page fault	4,587,164	87.5
Total	5,242,901	100.0

Table 1. Number of page faults during a 20GB sequential read on Fastswap. The local cache size is 2.5GB.

reclamation in the context of the page fault handler. Even though Fastswap reduces the reclamation time through a dedicated background kernel thread, not all reclamation work is offloaded to the thread, and reclamation still takes up a significant portion (29%) of the total page fault handling latency. This result indicates that in common cases (Average in Figure 1), reducing the software costs is essential to improve page fault handling latency.

Cost of IO amplification. When an application accesses a small object (less than 4 KB) in remote memory, a typical paging subsystem fetches the entire 4 KB page, causing IO amplification. To reduce IO amplification, AIFM fetches only accessed objects in a page using application-level semantics. SemSwap [15] tracks hot objects using user-level semantics and consolidates them onto a single physical page. While such fine-grained accesses effectively reduce the network bandwidth consumption and latency, we ponder the seriousness of the IO amplification; How severely does the object size affect the access latency? We run a simple experiment to measure the RDMA latencies of a range of object sizes between two nodes. Both nodes were configured as Ubuntu 18.04, and we used `libibverbs` in order to read and write objects. The results are in Figure 2. From the figure, we note that the RDMA latency for fetching a 4 KB page imposes only 0.6 μs extra delay compared to fetching a 128 B object; that is, even though the IO amplification increases network bandwidth consumption, it is not a major contributing factor in fetching delay.

3.2 Cost of Prefetching

Prefetching is a common yet integral mechanism to hide remote access latency. The swapping subsystem in Linux "offers a backup on disk for unmapped pages" [12], and

Fastswap, Infiniswap, and Leap all extend the swapping system to replace a local disk with remote memory. The Linux swap system has an intermediate buffer called swap cache to aggregate pages for swap-ins and swap-outs. Accesses to those pages in the swap cache result in minor page faults, and accesses to those not in the swap cache result in major page faults. We analyze the numbers of major and minor page faults in 20 GB sequential read on Fastswap and present the results in Table 1. Linux's default prefetching policy is readahead, and the given workload of sequential read results in a high hit ratio for pages in the local swap cache. Only 12.5% of pages incur major page faults, and the majority of 87.5% are minor page faults serviced from the swap cache. Latency from minor page faults is far smaller than major page faults, but the sheer number of such faults imposes overhead on the existing paging-based systems.

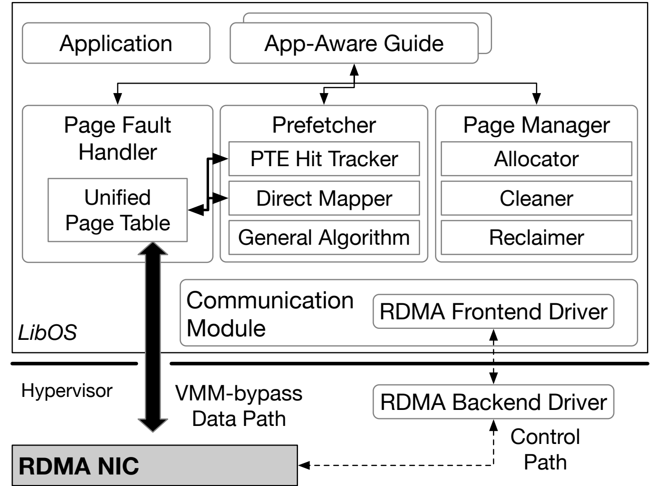


Figure 3. DiLOS' computing node overview

3.3 Our Approach: Specialized Kernel with LibOS

The analysis in the previous subsection reveals the overhead of the general-purpose design of the paging subsystem. Fastswap, Infiniswap, and Leap leverage the existing swap subsystem for ease of development, but the approaches come at a cost: the high cost of page fault handling and prefetching. However, the cost is not fundamental. We claim that *the kernel paging-based approach can overcome the overhead by specializing the kernel data-path for memory disaggregation.*

To demonstrate our claim, we revamp the existing paging mechanism to hide network latency to access remote memory and cut unnecessary software costs as well as to build a specialized prefetching mechanism. Although the general-purpose OSes have room for specialization, we observe that using Library OS (LibOS) is the best fit for our goal.

First, because a LibOS is linked to applications directly, each application has an exclusive LibOS kernel. This design translates to the ease of specializing operating system for memory disaggregation. Using the simplified codebase of a LibOS, we can build a more optimized data-path for abstracting remote memory than the traditional kernel. There are many examples of specializations using LibOS: ClickOS and MiniCache for NFV [35, 48], LightVM and ukvm for instant booting [47, 74], Haven and Graphene-SGX for enclave [6, 14], and HermitCore for HPC [40].

Second, by the design of LibOS' single address space memory layout and zero overhead system calls, LibOS can naturally integrate domain-specific hints to improve performance for memory disaggregation with negligible costs. We explore two opportunities: i) LibOS exploits application-level semantic hints and accelerates prefetching of remote pages (§4.3). ii) LibOS can use states of allocated and freed objects from the user-level memory allocator (*e.g.*, libc malloc) to exclude freed objects when fetching a remote page, saving the bandwidth consumption (§4.4).

Third, unlike the user-level approaches (e.g., AIFM), using a LibOS does not require modifications of existing applications and provides binary compatibility with POSIX APIs such as `malloc` and `free`.

4 System Architecture of DiLOS

DiLOS aims to rebuild a memory subsystem specialized for memory disaggregation. Unlike prior works [2, 23, 49], DiLOS does not rely on complex general-purpose kernel subsystems. Instead, it has a paging subsystem, which is simple yet tailored for a low page fault handling overhead. To minimize the overhead, the paging subsystem uses a *unified page table* as a replacement for Linux's swap cache. It shortens the code path between hardware exception and network IO and reduces the number of minor page faults. The subsystem also eagerly evicts local pages in the background to prevent page fault handling from being lagged from the reclamation.

We build DiLOS on a unikernel (LibOS) to take advantage of its lightweight architecture. DiLOS consists of four key components running on a computing node and a memory node. The four components are a page fault handler (§4.2), a page prefetcher (§4.3), a page manager (§4.4), and a communication module (§4.5). This section begins with the design overview of DiLOS (§4.1) and elaborates on its four key components. DiLOS' page prefetcher and page manager sections also introduce two app-aware guides to augment DiLOS, providing additional performance improvements and saving network bandwidth.

4.1 Design Overview

Figure 3 shows the overview of the computing node. An application and the LibOS run in a single address space and interact via POSIX APIs. The LibOS runs as a guest operating system, and the hypervisor offers virtualized interfaces, the frontend and backend drivers for network IO. The arrows in

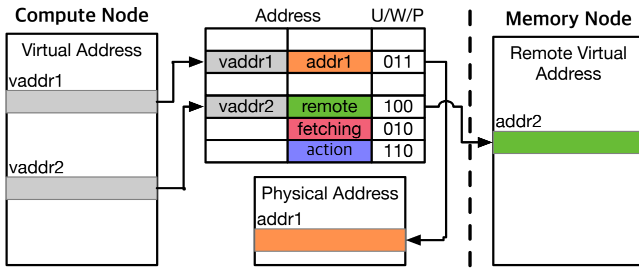


Figure 4. Unified Page Table

Figure 3 represent the interaction between the four key components of a DiLOS computing node. The guides next to the application represent app-aware pluggable modules. They are compiled code (*e.g.*, shared library binary) that enhances DiLOS performance based on app-specific prefetching and page reclaim implementation. The guides do not require modifications of the main applications, working as a third-party library to modify the default behaviors of DiLOS.

Unified page table. At the heart of DiLOS' paging subsystem lies the unified page table. It has a compact form representing the memory space for both local DRAM and remote memory without using the swap system or the swap cache. Figure 4 shows how the unified page table works. The unified page table complies with the hardware page table format (*e.g.*, the Intel four-level page table), and each page table entry (PTE) has four *DiLOS tags*: `local`, `remote`, `fetching`, and `action`. They are identified by the three least significant bits (user, write, and present bits) in the PTE. When the present bit is set, the PTE is local, and the hardware MMU translates a virtual address to a physical address (orange). For other cases, DiLOS uses the `remote` (green) and `fetching` (red) PTEs to mark their page as remote and fetching, respectively (§4.2). The `action` PTE (purple) embeds action data used to implement a page fault handling guide. If the PTE value is `action`, the DiLOS page fault handler calls the guide provided by the application developers. The guide refines the behavior of the DiLOS page fault handler. On behalf of the page fault handler, the guide issues custom prefetching requests (§4.3) and fine-grained fetching requests to save network bandwidth (§4.4).

4.2 Page Fault Handler

The main goal of the DiLOS page fault handler is to reduce the page fault handling latency. The key idea is to shorten all necessary code paths before making an asynchronous network request. The existing kernel paging-based systems have to go through the complex swap subsystem code causing high overhead for managing extra data structures (*e.g.*, the Linux swap cache). After allocating memory for the swap cache and its data structure, they issue an RDMA request. In contrast, the DiLOS' page fault handler checks only a single data structure before making an RDMA request, the unified page table. DiLOS encodes all information needed for memory disaggregation into the page table.

When an application accesses pages not in the local cache (whose present bit is zero), a page fault occurs. DiLOS' page fault handler checks the PTE value and handles the page fault depending on the DiLOS tag in the unified page table. If the PTE value indicates `remote`, the handler changes the value to `fetching` and makes an RDMA request to obtain its remote page. During the fetching, if a page fault handler in the other CPU core reads the fetching value, it just waits until the PTE value changes, preventing duplicated fetching requests from multi-threads. After completion of the fetching, the page fault handler maps the fetched page to the page table.

4.3 Page Prefetcher

DiLOS does not use the swap cache to store fetched (or prefetched) pages but maps them directly into the page table; All fetched and prefetched pages are mapped into the unified page table immediately, preventing minor page faults caused by the swap cache.

However, this design poses a challenge for implementing prefetching algorithms. Prefetch algorithms need the hit ratio and the access history of prefetched pages, which used to be provided by minor page fault statistics in the swap cache. Here, DiLOS takes an alternative approach to obtain the hit ratio and access history; it deploys an extra service called a PTE hit tracker. Upon prefetching, the PTE hit tracker scans accessed bits of prefetched PTEs and collects the result to calculate the hit ratio and access history.

Prefetching and hit tracker work do not incur extra latency to DiLOS page fault handling because they happen while the page fault handler waits for fetching a 4KB page. We observe that 2-3 μs (4 KB page fetching in Figure 1) is enough time to run the hit tracker and issue an asynchronous prefetch request, effectively hiding their latency to the window of a 4KB RDMA request.

DiLOS has two default general-purpose prefetchers: Linux's readahead prefetcher [28] and Leap's majority trend-based prefetcher [49]. Also, DiLOS' prefetcher has an event-based programming interface for guides. It supplies prefetching information (fault address, hit ratio, and access history) to an app-aware guide, and the guide provides prefetching code to augment the default prefetchers.

Guide for app-aware prefetching. For applications whose memory access patterns are irregular, general-purpose prefetching algorithms do not perform well in predicting the next pages. For example, when an application traverses a linked list, its memory access pattern is a sequence of pointer indirections (called pointer-chasing). In such an access pattern, the past access history does not provide meaningful information for the next page, and general-purpose prefetchers based on the history may perform as badly as none. Understanding

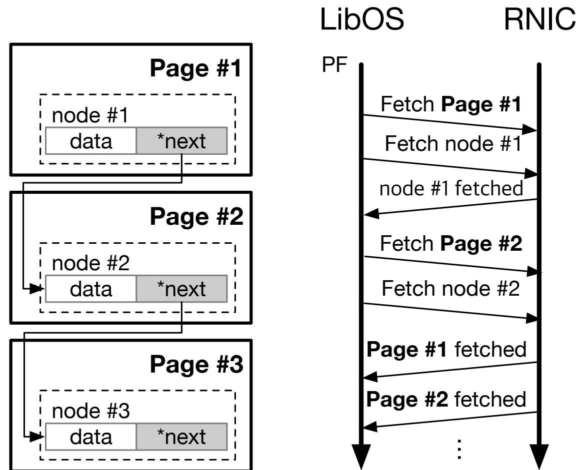


Figure 5. Memory layout of a linked list (left) and prefetching with an app-aware prefetcher guide (right)

the application semantics and devising a prefetching strategy is the necessary next step accordingly.

DiLOS showcases the design of an app-aware prefetcher to address the problem. Let us begin with a linked list as shown on the left side of Figure 5. Each node in the linked list is on a different page. During the list traversal, an application accesses Page #1, Page #2, and Page #3 in sequence. In order to prefetch the correct next page, the prefetcher must garner the information from the next pointers in the nodes. To prefetch the next page after Page #1, the prefetcher has to wait until Page #1 arrives. However, the required information is just the `*next` pointer value of node #1. What if the prefetcher issues a subpage prefetching for node #1 right after the fetching request for Page #1 has been issued? As shown on the right side of Figure 5, the subpage prefetching request may deliver node #1 ahead of Page #1 being fetched via a separate queue; thereby, the prefetcher gains knowledge about the next page to fetch quickly. The subpage is discarded after a page fetch for Page #2 has been issued.

In general, given the application semantics and the memory layout, an app-aware prefetcher guide plays the role of information conveyor from the application to the paging subsystem. The application dictates via prefetcher guide APIs how the prefetcher should work on behalf of the application. Application developers provide the guide as a compiled third-party binary so that it can be loaded with the existing application binary. If an app-aware prefetcher is not provided, DiLOS runs its default prefetcher.

4.4 Page Manager

Page reclamation during page fault handling takes a large portion of total latency in Fastswap as shown in Figure 1. DiLOS' page manager hides the page reclamation latency to the window of fetching a remote 4KB page via background and eager eviction of the least frequently used pages. When the page fault handler needs a new page, the *allocator* in the page manager provides a free page in the local DRAM. When the local DRAM becomes full, the page manager performs page reclamation to free pages. To avoid page reclamation executing in the critical path, the reclamation happens on a background thread and always keeps a few free pages by eagerly evicting the local cache. The reclamation process is done by two modules: a *cleaner* and a *reclaimer*. The allocator inserts all newly allocated pages into an LRU list. The cleaner periodically scans the LRU list to find dirty pages whose dirty bit is marked. Then, it writes them back to the memory node and clears the dirty bits (cleaning). When the system is under memory pressure, the reclaimer evicts the least frequently accessed clean pages according to the clock algorithm.

Guided paging for bandwidth reduction. Similar to the page fault handler and the prefetcher, DiLOS' page manager has an API for app-aware guides. To demonstrate the usage and usefulness of the API, we design guided paging that reduces network bandwidth.

The paging-based memory disaggregation systems always use 4KB pages for communication. As discussed in §3.1, the IO amplification due to the unit of 4KB page has a limited effect on page fault handling latency. However, it still wastes network bandwidth. In this case, transferring only used areas (i.e., live objects) of pages reduces the network bandwidth. To achieve it, DiLOS presents a new paging mechanism, which exploits vectorized (or scatter/gather) RDMA requests to fetch and evict only used areas on a page. The guide uses allocation information in an application-level memory allocator (`malloc`) to identify the used areas. This guided paging is done transparently to existing applications.

The guided paging utilizes the action PTE in the unified page table. At a high level, the app-aware guide encodes the positions of the live objects within a page to its PTE value, and then the page fault handler and reclaimer use the PTE value to fetch and reclaim the live objects only. During the page reclamation, the cleaner calls its app-aware guide. Then, the guide identifies and returns which chunks in a page are currently used by reading the allocator's memory layout. The DiLOS' user-level memory allocator maintains per-page allocation bitmaps to represent the live objects. The cleaner writes the only used area in an evicted page to the memory node with a vectorized RDMA request instead of writing the entire page contents. After completion of the RDMA request, the cleaner logs the request's vector, and then the reclaimer evicts the page by updating its PTE to an `action` PTE. Later, when the evicted page is accessed, the `action` PTE contains vector information used for fetching the page.

The guide helps DiLOS paging systems to eliminate unused areas when evicting and fetching a page, effectively saving network bandwidth. The benefit of the guide is using only allocator semantics, applicable to all applications and runtime without requiring application semantics.

4.5 Communication Module

DiLOS' communication module aims to mitigate the source of unexpected delays in RDMA connections, such as head-of-line blocking or lock waiting. To hide the delays, the communication module handles requests from the multiple modules in a shared-nothing manner. For example, the page fault handler's requests must not be blocked by other low prioritized requests from a prefetcher or a manager (head-of-line blocking). To prevent this situation, the communication module assigns per-module queues to each core. This way, regardless of the core that it runs, any paging module disables preemption and gains blocking-free access to an RDMA queue. Also, multiple threads running on a single core share the queue without expensive locking.

The communication module also assigns per-core RDMA queues for app-aware guides. The guides may employ their own subpaging mechanisms and need separate queues from the paging modules. The per-core queues for a guide guarantee blocking-free access to subpaging requests.

5 Implementation

We have built DiLOS based on the OSv unikernel version 0.55 [32]. DiLOS' core part is written in 4,454 LoC [1]. We have also modified 322 LoC of OSv to link the core part and 442 LoC of QEMU (version 4.2.1) to implement the RDMA backend driver. DiLOS' memory node consists of 312 LoC.

Prefetchers and guides. DiLOS has two general-purpose prefetchers based on readahead [28] and majority trend [49]. We have also implemented app-aware guides for the prefetcher and the allocator. The prefetcher guide uses Redis [61]'s data structure layouts to predict memory access patterns, and their detailed implementation is described in §6.3. The app-aware allocator guide of DiLOS is based on Microsoft's mimalloc [42], which is a simple yet high-performance memory allocator. DiLOS' allocator tracks subpage usages via bitmaps. The prefetcher consists of 275 LoC, and the allocator has 951 LoC of modifications in the mimalloc.

Memory node. DiLOS uses one-sided RDMA for communication. A server process in the memory node handles setup requests from the computing node and registers its memory region to its RDMA NIC (RNIC). After that, the RNIC serves all read and write RDMA requests from the computing node. Lastly, the memory node uses huge TLB pages for the memory region. Huge pages allow the whole RNIC page table to fit in the RNIC's cache and reduce host memory accesses for page table walks.

Low-latency RDMA driver. The RDMA driver available at the moment of our system development lacked support for OSv. We have built our own RDMA driver. We separate the data-path from the control-path in DiLOS. Our focus in driver development is a fast data-path and an easy-to-implement control-path on the virtualized platform.

Using a virtualized RDMA driver causes significant overhead for the data-path due to extra copying of payloads from LibOS (guest OS) to the host driver (hypervisor) and VM exits. To avoid this overhead, DiLOS bypasses the hypervisor and directly communicates with RNICs. RDMA's data-path involves two steps (write, for example): putting data in the RDMA memory region and issuing commands to an RNIC through the MMIO region in user-space. Then, the RNIC reads the user-space data using DMA. To communicate with RNICs, DiLOS' host driver exposes its RDMA and MMIO regions to LibOS' (single) memory address space and populates the mapping table of the address space in the RNICs. This way, LibOS accesses RNICs directly, and the driver's data-path requires neither expensive VM exits nor data copying, exhibiting performance close to native RDMA drivers.

To isolate data-path among VMs, DiLOS' driver uses RDMA's protection key mechanism. In RDMA, each memory region is associated with its own protection key, and RNICs allow accessing the memory region only when a proper protection key is provided. Therefore, a malicious LibOS cannot access arbitrary memory regions assigned to other LibOSes even if they share the same RNIC.

For the simple implementation of the control-path, our RDMA driver reuses the host-side device driver. Our driver passes control-path requests to the backend driver in the hypervisor using `virtio`. Then, the backend driver adds modifications to the requests (*e.g.*, to populate the address mapping table in an RNIC, the driver translates all addresses to LibOS' addresses) and issues them to the host-side device driver. Since `virtio` requires VM exits, the control-path is slower than running in native RDMA drivers. However, DiLOS and other memory disaggregation systems use the control-path only once at the initialization stage to establish a connection between the computing node and the memory node. Therefore, the high cost of the control-path does not affect the overall performance of memory disaggregation systems, and we have decided to trade off the performance for simple implementation.

Our design choices in the driver implementation are identical in spirit to VMM-bypass [43], MasQ [25], and HyV [54]: directly mapping data-path resources to VMs' memory region. We have also considered PVRDMA [56], but we have opted out. Its driver exits the VM upon every request, delivering limited performance.

Our RDMA driver has additional IO optimization to gain low latency close to the host driver. Mellanox RDMA devices offer the BlueFlame (or WQE-by-MMIO) feature, which enables transferring RDMA commands over MMIO, not DMA. BlueFlame is an essential feature for low latency [29]. However, the vanilla OSv lacks support for a write-combining buffer that BlueFlame uses for the efficient MMIO. We have

[1] We use `SLOCCount` for the original code and `diff` for modifications.

modified the OSv to support the write-combining buffer and enable the BlueFlame feature on DiLOS.

Compatibility layer. DiLOS uses two types of memory APIs for disaggregated memory and local-only memory. Both types create and destroy virtual addresses in the unified page table, but pages allocated by disaggregated memory APIs are migrated to the memory node. To use memory disaggregation, the application has to use `ddc_malloc` and `ddc_free` functions. Internally, the function uses the `mmap` call with the `MAP_DDC` option, indicating memory disaggregation is enabled. The `mmap` call returns a virtual address whose page faults are handled by DiLOS' page fault handler.

To provide binary compatibility, DiLOS has a custom ELF loader to replace original APIs with DiLOS APIs. In the application loading stage, the ELF loader patches all `malloc` and `free` calls in the application's symbol table with corresponding DDC APIs. When implementing guides, the ELF loader provides hooking interfaces of an application binary. Some guides often use the hooking interface to collect necessary application information. For example, in §4.3, the prefetcher has to know the position of the traversing node to fetch the first node (node #1). Thereby, the prefetcher hooks the list traversing code and tracks the position of the current node.

5.1 Discussion

Applying features of DiLOS to general-purpose OSes. The paging subsystem design of DiLOS does not have inherent dependence on LibOS systems. In theory, it should be applicable to general-purpose OSes. For example, the swap cache in the Linux kernel may be replaced with a unified page table, a direct mapper, and a hit tracker. Linux may utilize the eBPF subsystem [31] to implement app-aware guides. Fastswap may adopt DiLOS' page manager design to hide its reclamation time in the page fault handler. However, unlike LibOS, general-purpose OS' mode switching and multi-applications model limit the performance of the paging subsystem design. DiLOS employs LibOS to show the performance without the overheads.

Applying DiLOS to disk-based swapping. Since the paging-based memory disaggregation originated from disk-based swapping, the DiLOS' design choices (*e.g.*, shortening the code path between exception and I/O) can improve the disk-based swapping performance also. However, traditional block devices (HDDs and SSDs) are much slower than far memory using RDMA. Accordingly, the I/O will be the dominant overhead hiding performance improvements of DiLOS' design. Modern NVMe drives provide enough performance to be used for far memory [39]; thereby, DiLOS' design would be valid for NVMe drives.

Supporting multiple nodes and fault tolerance. Under the current DiLOS implementation, a computing node only supports one memory node, just as in Fastswap and AIFM. We have not included recovery from a memory node failure in this work, yet an asynchronous storage backup mechanism [23] or erasure-coding-based replication [77] is one candidate approach for fault tolerance. Extending DiLOS to support multiple memory nodes for replication or sharding is a future research direction.

5.2 Limitations

Since we have built DiLOS based on OSv, DiLOS inherits the same limitations as OSv. Though OSv is POSIX-compatible and supports unmodified Linux ELF binaries [33], it lacks multi-processing APIs such as `fork()`. However, most modern applications do not use the `fork()` system call [5], and DiLOS supports multi-threading alternatives (`pthread`) for multi-core support.

The current DiLOS implementation does not support live migration. Because it bypasses its hypervisor and directly stores states (*e.g.*, registered buffers and queue pairs) in the NIC, migrating the NIC-internal states is challenging. A recent study proposes a modification to the RDMA communication protocol for migrating NIC-internal states [55]. Once the proposed modification becomes standard, supporting the live migration of DiLOS should be straightforward.

6 Evaluation

At the core of our DiLOS lie the following design choices.

- We have employed a unified page table design rather than Linux's swap cache design.
- In our revamped paging subsystem, we have introduced novel page fault handler, prefetcher, and page manager designs that minimize page fault handling overhead.
- The communication module embraces a share-nothing principle in order to prevent unexpected delays.
- App-aware guide fine-tunes performance with insights from application semantics.

How does our DiLOS fare in comparison to existing systems? In this section, we answer the question in the following order. First, we evaluate the cost of paging in terms of throughput and latency against Fastswap (§6.1). The workload we use here is sequential read and write. Second, we use simple benchmarks and real-world applications to compare performance against Fastswap and AIFM, as AIFM delivers the best performance among existing systems (§6.2). Finally, we demonstrate the performance improvements that app-aware guides deliver (§6.3).

Testbed. We set up a computing node and a memory node connected with a 100GbE RoCE cable. Each node has an Intel Xeon CPU E5-2670 v3 2.30GHz, DDR4 RAM, and a Mellanox ConnectX-5 EDR + 100GbE card (CX556A). DiLOS use QEMU 4.2.1 and Mellanox OFED 5.0 on top of Ubuntu 18.04 and Linux kernel 4.15. Both Fastswap and AIFM run on Ubuntu 18.04 and Mellanox OFED 4.6 for compatibility, but their Linux kernel versions are 4.11 and 5.0, respectively. We

	Read	Write
Fastswap	0.98	0.49
DiLOS with `no-prefetch`	1.24	1.14
DiLOS with `readahead`	3.74	3.49
DiLOS with `trend-based`	3.73	3.49

Table 2. Throughputs of sequential read and write (GB/s).

confirm that both OFED versions of 4.6 and 5.0 show almost identical performance in the `perftest` utility [58]. To vary available memory in computing nodes, we use QEMU's `m` option for DiLOS, LXC container for Fastswap, and `kCacheGBs` constant for AIFM. Throughout §6, memory nodes use 2MB huge pages and run without offloading features for a fair comparison. We run the same workload five times for each experiment and report the average.

Presentation. In all experiments, we label results from DiLOS without a prefetcher, with the readahead prefetcher [28], and with the Leap's trend-based majority prefetcher [49] as `no-prefetch`, `readahead`, and `trend-based`, respectively. DiLOS with the app-aware guided prefetcher or guided paging is marked as `app-aware`.

6.1 Cost of Paging

The goal of this subsection is to evaluate the basic performance, namely, throughput and latency of the paging subsystem under sequential read and write. The workload first allocates and populates 20GB of memory and then reads or writes the region with 4KB strides. We use 2.5GB (12.5%) of local cache for all systems.

Throughput from sequential read and write. Table 2 shows read and write throughputs of DiLOS and Fastswap. DiLOS shows superior performance compared to Fastswap in all configurations. DiLOS without prefetchers shows 26% and 134% higher read and write throughputs than Fastswap. With prefetchers, DiLOS further hides network and hardware latencies. The readahead prefetcher enhances DiLOS' read and write throughput by about 3.0× and 3.1×, respectively. We presume the drastic performance improvement mainly comes from two design choices in DiLOS; reduced latency in page fault handling and smaller numbers of minor page faults while prefetching. Next, we analyze the latency and minor page faults in detail.

Latency breakdown. Figure 6 shows the latency breakdown of the page fault handler in DiLOS and Fastswap. We added a bar graph for Fastswap without reclamation in order to highlight the increased latency from direct reclamation. The main reduction of DiLOS comes from page allocation and reclamation. The reduction in page allocation is as expected, for DiLOS got rid of the swap cache. Fastswap is designed to hide reclamation time to asynchronous RDMA requests using *offloaded reclaim*, but not all reclamation are offloaded. DiLOS, on the other hand, succeeds in completely hiding the reclamation time, and thus we see no reclamation time in Figure 6. Overall, DiLOS reduces the page fault handling latency by about 49%.

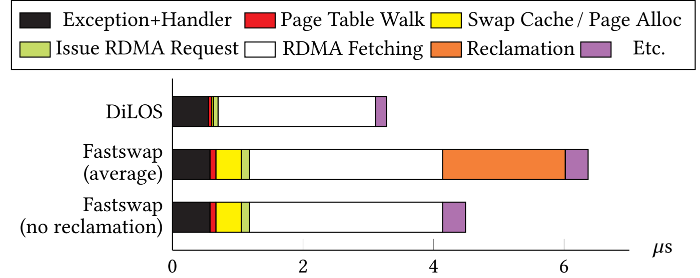


Figure 6. Latency breakdown of DiLOS and Fastswap during sequential read. Prefetch is turned off for both.

	Major	Minor	Total
Fastswap	655,737	4,587,164	5,242,901
DiLOS with `no-prefetch`	5,242,880	-	5,242,880
DiLOS with `readahead`	655,358	3,167,233	3,822,591
DiLOS with `trend-based`	655,356	3,252,667	3,908,023

Table 3. Number of page faults during sequential read.

Number of page faults. Table 3 shows the numbers of major and minor page faults while executing the sequential read workload. Without prefetching, DiLOS has as many major faults as Fastswap's major and minor combined. With prefetching enabled, DiLOS shows similar numbers of major faults to Fastswap. DiLOS' prefetchers update page table entries directly and, in turn, incur about 25% fewer minor page faults than Fastswap.

The results confirm that faster page fault handling and fewer minor page faults are the main reasons that DiLOS performs better than Fastswap.

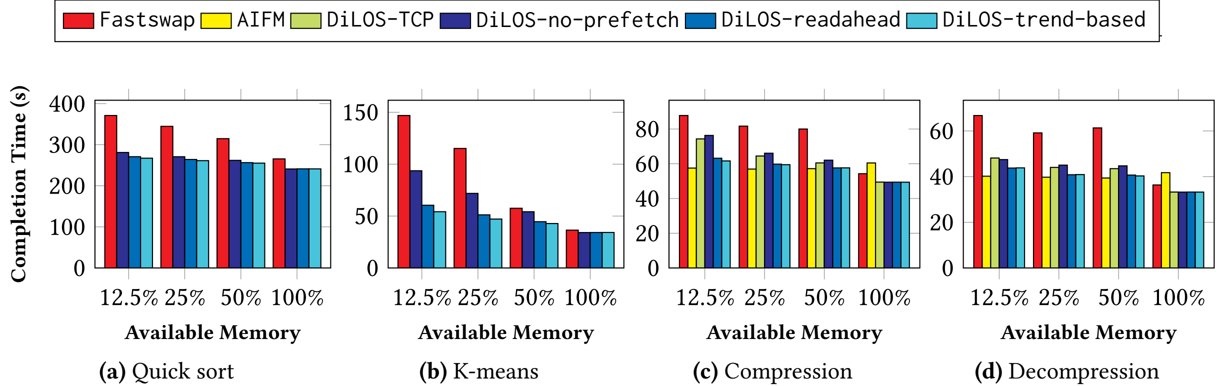
6.2 Performance under Diverse Workloads

In this subsection, we use simple benchmarks (quick sort, k-means clustering, and compression) and real-world applications (data analytics, graph processing, and in-memory key-value store) to compare DiLOS to Fastswap and AIFM.

For a fair comparison with AIFM, which uses TCP, we have added 14,000 cycles of delay after each RDMA completion[2]. In these cases, we use the Linux's readahead prefetcher and label it `DiLOS-TCP`.

Since AIFM requires C++ programming language and application porting, we compare DiLOS to AIFM only with applications from their publicly available code: Snappy compression (Figures 7(c) and 7(d)) and DataFrame (Figure 8).

[2]In our testbed, we measured AIFM's TCP and RDMA 4KB read performance and saw that AIFM's TCP is 14,000 cycles slower than RDMA.



(a) Quick sort **(b)** K-means **(c)** Compression **(d)** Decompression

Figure 7. Completion time of simple benchmarks. Lower is better.

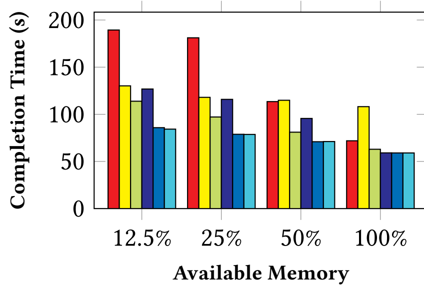
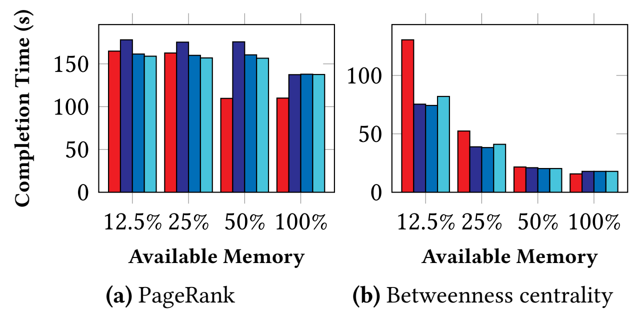


Figure 8. Completion time of the NYC taxi workload on DataFrame. Lower is better.



(a) PageRank **(b)** Betweenness centrality

Figure 9. GAPBS processing time. Lower is better.

Simple benchmarks. We begin with quicksort, k-means clustering, and compression workloads. The quicksort workload allocates a vector of 2048M random integer numbers (total 8GB) and sorts them with C++'s `std::sort` function. The k-means clustering workload uses Scikit-learn [52] to classify randomly generated 15M integers into 10 clusters. We use Python 3.6, Numpy 1.13, and Scikit-learn 0.19. They are the default versions provided by Ubuntu 18.04. The last workload performs in-memory compression and decompression using Snappy library version 1.1.8 [21]. Just as in AIFM work [60], we use sixteen 1GB files for compression and thirty 0.5GB files for decompression.

Figure 7(a) shows the quicksort result. Fastswap's completion time increases by 39%, as the local cache size decreases from 100% to 12.5%. DiLOS fares better with only a 12% increase in completion time. With 12.5% local memory, DiLOS shows up to 1.39× better performance than Fastswap.

Figure 7(b) shows the completion time of the k-means algorithm. Reducing the local memory has a bigger impact on the completion time than in the quick sort case. The k-means clustering algorithm exhibits much irregular memory access patterns, stressing the slow page reclamation for evicting pages than the quicksort workload; therefore, DiLOS shows much higher performance improvement in the k-means workload. With 608MB of local cache (12.5%), DiLOS boosts the performance of the k-means algorithm up to 2.71× compared to Fastswap.

Figures 7(c) and 7(d) show performance from Snappy compression and decompression. Since both workloads have sequential memory access patterns, prefetchers hide network latency. When the available local cache size is limited (12.5%), AIFM has better performance than the other systems. Compared to AIFM, DiLOS with prefetchers has only a 7-9% slowdown, and DiLOS with TCP has a 17-23% slowdown, while Fastswap has a 35-40% slowdown. AIFM's good performance is due to its multi-threaded prefetcher that runs in background and enables almost perfect overlapping of computation and networking. However, when the computing node has enough memory (*e.g.*, 50% and 100%), AIFM tends to be similar to or slower than DiLOS because, when dereferencing pointers, AIFM needs to execute extra instructions to check whether accessing objects are in local or remote memory. While AIFM requires application modifications, DiLOS delivers reasonable performance without application modification.

Data analytics. For real-world applications, we have listed data analytics, graph processing, and in-memory key-value store applications. Data analytic libraries are commonly used in data science and machine learning. We use the C++

DataFrame library [50] available in AIFM's repository [59]. We also employ the same data-set offered by AIFM, the New York City taxi trip analysis workload [30]. In our testbed configuration, the workload requires 40GB of memory, and thereby we limit the available memory on the computing node to 20GB (50%), 10GB (25%), and 5GB (12.5%).

Figure 8 shows the total completion times of all three systems. When the local cache in a computing node is large enough (100%), AIFM has 50-83% slower performance than the other systems. DiLOS, even with the TCP emulation delay, has 14% better performance than AIFM, and with RDMA, the gap increases up to 54%.

The DataFrame workload has spatial locality, if not perfect as in sequential read workload. The completion times of AIFM and DiLOS increase slightly as the available memory decreases, while that of Fastswap more than doubles. The results demonstrate that our paging-based memory disaggregation delivers comparable performance to AIFM for the NYC taxi workload.

Graph processing. To evaluate DiLOS on a multi-threaded workload, we use GAPBS (GAP Benchmark Suite) [7] of the latest version 1.4. The workload runs two graph processing algorithms—PageRank (PR) and betweenness centrality (BC)—with the Twitter data-set [37]. The total working set of the workload is 17GB. The data access pattern of betweenness centrality is more random than PageRank, as it traverses one more indirection through tables. For all experiments, the number of threads is set to 4.

Figures 9(a) and 9(b) show GAPBS processing time of Fastswap and DiLOS. When a computing node has enough memory (50% and 100%), DiLOS has a longer completion time of PageRank than Fastswap due to OSv's synchronization overhead. As OSv is not as mature or widespread as Linux, we believe this performance penalty to be alleviated over time. Under common memory-constrained settings (*e.g.*, 12.5%) in DDCs, DiLOS shows up to 76% higher performance than Fastswap in betweenness centrality computation.

In-memory key-value store. In-memory key-value store applications use pointer-based data structures (*e.g.*, hash tables and linked lists), and they have highly irregular memory access patterns, impacting the effectiveness of a prefetcher. We use Redis [61], a popular in-memory key-value store, and run `GET` and `LRANGE` workloads using Redis's benchmark tool `redis-benchmark` [38]. We use Redis 6.2.4, the latest stable version at the time of our evaluation.

To represent a memory-intensive `GET` workload in a datacenter, we run a workload generator with a data-set similar to Facebook's photo-serving server [27]. Since the `GET` requests dominate most key-value stores [3], we focus on `GET` serving performance with three workloads of different data sizes. Before measurement, we fully populate the key-space of the Redis server with `4KB`, `64KB`, and `mixed`-sized data, respectively. The working set size is approximately 20GB. While `4KB` and `64KB` workloads consist of fixed-size data, the `mixed` workload has six equally distributed data sizes —4KB, 8KB, 16KB, 32KB, 64KB, and 128KB—which represent data sizes of more than 80% of objects in the Facebook's photo server. Finally, we send `GET` queries with random keys 4M, 250K, and 396K times (the same number as the key-space) with `redis-benchmark`.

The evaluation results from `GET` workloads are in Figures 10(a) to 10(c). When an object size is 4KB, the object fits into a single page, and thus the randomness of the next memory access increases. This weakens the effectiveness of a prefetcher, as shown in Figure 10(a). On the other hand, as the object size grows, the object stretches over multiple contiguous pages. Next pages to fetch become predictable in such a case, making prefetchers more effective - in the best case (`trend-based` on `64KB` workload), the throughput is 63% higher than `no-prefetch` (Figures 10(b) and 10(c)).

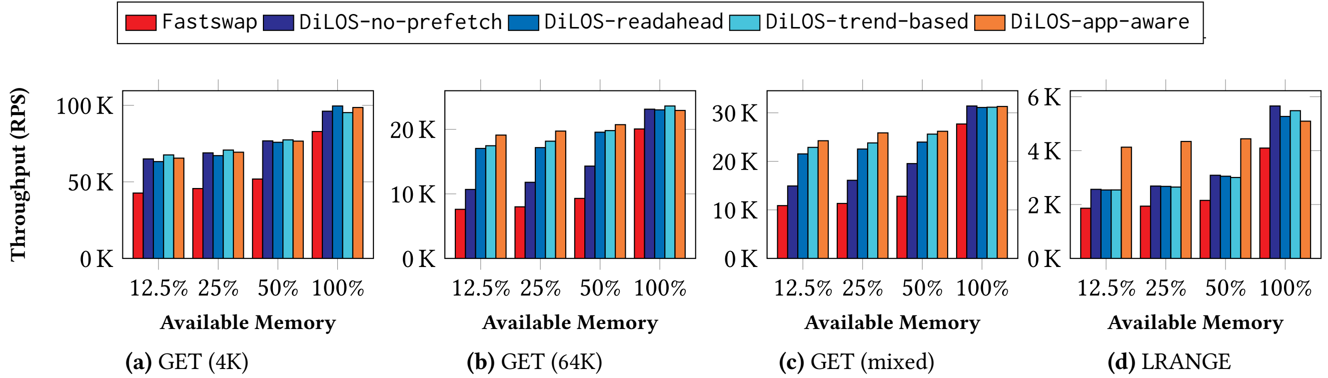
The `LRANGE` query returns a set of elements from a list. It is heavily used to deal with sequential data such as timelines or homepage feeds in social network services [26, 63]. We evaluate `LRANGE_100` performance from `redis-benchmark`, which retrieves the front 100 elements from a list. Vanilla `redis-benchmark` evaluates `LRANGE` query performance with only one list, which is not realistic in modern datacenters. We have modified the benchmark to populate and query 100 thousand separate lists. To populate these lists, we randomly pushed 20 million elements to lists so that each list contains 200 elements on average, using about 22GB of memory. Then we ran the `LRANGE` query 100K times.

For all configurations of available local memory, `GET` queries with different data sizes, and `LRANGE` queries for varying-length lists in Figure 10, DiLOS outperforms Fastswap. DiLOS, even without any prefetcher, has 1.37-1.52× higher throughput than Fastswap under memory-constrained conditions (12.5%). DiLOS with general-purpose prefetchers shows up to 2.51× higher throughput than Fastswap. General-purpose prefetchers improve DiLOS' performance further on `GET` workloads. DiLOS with `trend-based` enhances the Redis throughput by 1.63× from `no-prefetch`. However, these general-purpose prefetchers fare no better for `LRANGE` workloads where the memory access pattern is very irregular than in the case with no prefetching. Both prefetchers gain no performance gain over `DiLOS-no-prefetch`.

They are fundamentally sequential in their prefetching, diverging only in the amount of prefetching data and timing. For structured yet non-sequential data traversals, we need application-level information about the data layout.

6.3 Improvement from App-Aware Guides

In Figure 10(d), we have seen the limitations of prefetchers where data access is not sequential. The `LRANGE` query uses a quicklist data structure, which stores strings in a linked list of ziplists [66]. We have built a prefetching guide that traverses the quicklist and prefetches its elements in a way similar



(a) GET (4K) **(b)** GET (64K) **(c)** GET (mixed) **(d)** LRANGE

Figure 10. Redis's request handling throughput under GET and LRANGE workloads. Higher is better.

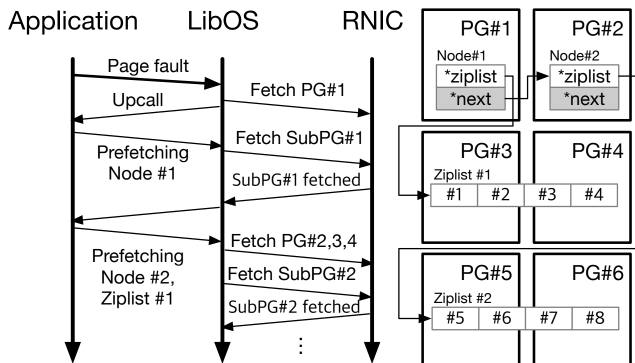


Figure 11. Quicklist prefetcher. PG and SubPG mean page and subpage, respectively.

	GET (mixed)		LRANGE	
	99th	99.9^{th}	99th	99.9^{th}
Fastswap	10.0	11.0	25.8	34.3
DiLOS with no-prefetch	6.2	7.6	18.0	20.8
DiLOS with readahead	3.0	4.0	18.0	20.8
DiLOS with trend-based	3.0	4.0	18.0	20.0
DiLOS with app-aware	3.0	4.0	14.6	18.4

Table 4. Tail latency of GET (mixed) and LRANGE workloads with 2.5GB local memory (millisecond).

to Figure 5. Figure 11 shows the details of our prefetcher for the LRANGE query quicklist. When a page fault for PG#1 takes place, the guide places subsequent prefetching calls. It first calls for a subpage prefetching to Node#1, and examines Node#1's ziplist. Then it issues page fetching commands for PG#2, PG#3, and PG#4 where the next node resides and the data of ziplist#1 occupies. Also, it issues a fetch for a subpage PG#2 for Node#2. Since the node's size is much smaller than a full page, the subpage arrives ahead of the full page from the first page fault. The prefetcher uses the node's values to fetch its ziplist (Ziplist #1) and the next node (Node #2).

App-aware guides are applicable not only in prefetching but also in network bandwidth reduction. Below we present an app-aware prefetcher for Redis and guided paging in an allocator for bandwidth reduction.

App-aware prefetcher for Redis. Above, we have presented a prefetching guide for the LRANGE query. We also have designed a guide for the GET query. To handle GET queries, Redis uses the Simple Dynamic Strings (SDS) library [62]. Redis's SDS consists of a header and data followed by a termination character. The length information is helpful for the prefetcher to decide the number of pages to prefetch. When a page fault occurs during a GET request, the prefetcher reads the header part first and uses its length information to fetch the exact number of pages.

The app-aware prefetcher for GET and LRANGE is written in only 275 lines of C code and compiled with the Redis source. It includes four handlers for subpage prefetching and four hooker functions for application information gathering. *Note that we need not modify the Redis main code for the prefetcher.*

In Figure 10 and Table 4, app-aware indicates the throughput and tail latency of DiLOS with the app-aware prefetcher. For GET workloads, the app-aware prefetcher performs on par with other prefetchers. For the LRANGE workload, the app-aware prefetcher outperforms the general-purpose prefetchers by 62% and reduces 99% tail latency by 18%. Overall, the app-aware prefetcher has 2.21× higher throughput and 28% lower 99% tail latency than Fastswap.

Guided paging for bandwidth reduction. We use Redis again to demonstrate the utility of the guided paging in bandwidth reduction as presented in §4.4. When Redis handles GET queries for objects in remote memory, we should avoid IO amplification and fetch only subpages of the objects.

The original mimalloc uses a list to track freed chunks. We modify the mimalloc code to use bitmaps to track freed chunks. When GET queries are issued, the modified mimalloc refers to the bitmaps and only fetches allocated subpages.

For the GET workload used in the evaluation, we use the SET operations to populate Redis's key-space (128 M keys with 128 bytes values) and the DEL operations to delete randomly about 70% of the key-space. The DEL operations leave

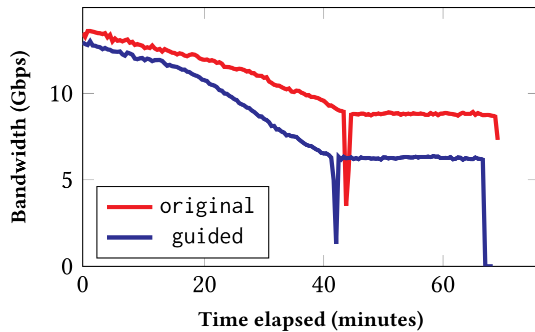


Figure 12. Bandwidth consumption during DEL (before 40 mins) and GET (after 40 mins) operations. Lower is better.

pages with free memory chunks. We limit the available memory to 2.5 GB, about 25% of memory in use after DEL operations. We have found that vectorized RDMA has a significant slowdown when its vector is longer than three. Therefore, the guide constructs a vector whose length is at most three.

Figure 12 shows network bandwidth consumption during DEL and GET operations. The guided allocator reduces bandwidth consumption by 12% for DEL on average and 29% for GET operations. During the DEL operations, page-internal fragmentations take place, and the vectorized paging kicks in to save network bandwidth.

7 Related Work

Memory disaggregation systems. Infiniswap first introduced kernel-based memory disaggregation using a swap device [23], and the following works [2, 39, 49] enhance the swapping performance via frontswap design [46]. Remote regions is a kernel-based memory disaggregation system, which uses a file abstraction represented via file (read/write) or memory mapping (mmap) [1]. LegoOS proposes a split kernel model, which disseminates conventional kernel features to loosely-coupled monitors running on each hardware[64].

On the other hand, user-level systems expose memory disaggregation APIs to applications. FaRM provides a lock-free RDMA API for fast access to remote memory [16], and CoRM extends FaRM to perform memory compaction [67]. Using FaRM or CoRM for disaggregated memory requires significant source code modifications of applications since they have custom APIs. Semeru [72], MemLiner [73], AIFM [60], and Carbink [77] preserve the language's standard APIs to reduce the modifications. Semeru and MemLiner implement memory disaggregation functions in JVM and allow the running of unmodified Java applications. AIFM and Carbink provide C++ standard-like far memory interfaces. SemSwap passes user-level semantics to the kernel to consolidate them onto the same physical pages [15].

Many memory disaggregation systems provide fault tolerance regardless of the kernel-based or user-level system. To address a remote machine failure, FaRM replicates the same data across multiple memory nodes [16]. Infiniswap writes replicated data onto the local block disk asynchronously [23]. Hydra [41] and Carbink [77] use erasure-coding to reduce memory overheads from replications.

Researchers also propose hardware designs for memory disaggregation. Shoal introduces a new circuit switch-based networking architecture to accommodate high-density disaggregated nodes [65]. CacheCloud discusses that a 400 Gb/s network fabric will allow hundreds of ns latency to access remote nodes, which is comparable to the latency between CPU and memory in a local node [69]. Aquila is a custom ASIC-based ultra-low latency datacenter network fabric which supports remote memory access as well as traditional traffic [20]. Kona proposes new hardware primitives for remote memory access at cache-line granularity [68]. DirectCXL enables remote memory access via load/store instructions without page faults via CXL interconnect [22].

Library operating systems and specialization. LibOSes have emerged in various domains. To take full advantage of hardware features, Dune [8] allows applications to directly perform privileged operations such as exception handling and virtual memory management. Arrakis [53] and IX [9] implement networking stacks in libraries to provide high I/O performance. Graphene focuses on running multi-process applications with a limited number of system calls for narrower attack surfaces [71], and Graphene-SGX runs Graphene on an enclave [14]. Unikernel is a branch of LibOS that adopts a single-address-space design and targets virtual machine environments [44, 45]. In succession to the first introduction of the unikernel, POSIX-compatible unikernels [32, 34, 36, 40, 51, 57] and language-based unikernels [4, 13, 18] have been proposed in the literature. LibOS has also acted as a mean for specializations in many contexts, including NFV [35, 48], instant booting [47, 74], enclave [6, 14], and HPC [40].

Besides LibOSes, general-purpose OSes are also evolving to enable kernel specialization. SPIN proposes new OS architecture to safely extend kernel features via type-safe language [10]. Exokernel allows applications to control hardware in specialized ways using low-level interfaces [17]. Linux also opens opportunities for safe specialization via eBPF [31]. BMC uses the eBPF subsystem to accelerate Memcached via pre-stack processing in the kernel [19]. XRP hooks NVMe driver codes via eBPF and offloads storage functions to the kernel [76].

8 Conclusion

In this paper, we review the costs of the kernel-based memory disaggregation systems and present DiLOS[3], an efficient paging subsystem based on LibOS for memory disaggregation. To hide the costs, it specializes the kernel data-path for

[3]This work extends the initial design from our workshop paper and adds detailed evaluations. Guided paging for bandwidth reduction is a new case study of specialization, previously not in the workshop paper [75].

memory disaggregation and uses user-space semantics to augment their behavior. DiLOS delivers higher performance than both the user-level approach (54%) and the kernel-based approach (124%) on a real-world workload. Compared with DiLOS' default version, DiLOS with app-aware guides shows 62% higher throughput on range queries and consumes 29% lower bandwidth on GET queries in Redis.

Acknowledgement

We thank our shepherd Yizhou Shan and anonymous reviewers for helpful comments. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2022R1A2C2009062) and Samsung Advanced Institute of Technology, Samsung Electronics Co., Ltd.

References

[1] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 775–787, Boston, MA, July 2018. USENIX Association.

[2] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can Far Memory Improve Job Throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[3] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[4] Fran. J. Ballesteros. Clive. https://lsub.org/clive/.

[5] Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. A fork() in the road. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 14–22, New York, NY, USA, 2019. Association for Computing Machinery.

[6] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with Haven. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 267–283, Broomfield, CO, October 2014. USENIX Association.

[7] Scott Beamer, Krste Asanović, and David Patterson. The GAP Benchmark Suite, 2017.

[8] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 335–348, Hollywood, CA, October 2012. USENIX Association.

[9] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, October 2014. USENIX Association.

[10] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, page 267–283, New York, NY, USA, 1995. Association for Computing Machinery.

[11] Sergey Blagodurov, Mike Ignatowski, and Valentina Salapura. The Time is Ripe for Disaggregated Systems. https://www.sigarch.org/the-time-is-ripe-for-disaggregated-systems/.

[12] Daniel P Bovet and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management.* " O'Reilly Media, Inc.", 2005.

[13] A. Bratterud, A. Walla, H. Haugerud, P. E. Engelstad, and K. Begnum. IncludeOS: A Minimal, Resource Efficient Unikernel for Cloud Services. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 250–257, 2015.

[14] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-sgx: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 645–658, Santa Clara, CA, July 2017. USENIX Association.

[15] Siwei Cui, Liuyi Jin, Khanh Nguyen, and Chenxi Wang. Semswap: Semantics-aware swapping in memory disaggregated datacenters. In *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '22, page 9–17, New York, NY, USA, 2022. Association for Computing Machinery.

[16] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, April 2014. USENIX Association.

[17] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, page 251–266, New York, NY, USA, 1995. Association for Computing Machinery.

[18] Galois, Inc. The Haskell Lightweight Virtual Machine. https://github.com/GaloisInc/HaLVM.

[19] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. BMC: Accelerating memcached using safe in-kernel caching and pre-stack processing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 487–501. USENIX Association, April 2021.

[20] Dan Gibson, Hema Hariharan, Eric Lance, Moray McLaren, Behnam Montazeri, Arjun Singh, Stephen Wang, Hassan M. G. Wassel, Zhehua Wu, Sunghwan Yoo, Raghuraman Balasubramanian, Prashant Chandra, Michael Cutforth, Peter Cuy, David Decotigny, Rakesh Gautam, Alex Iriza, Milo M. K. Martin, Rick Roy, Zuowei Shen, Ming Tan, Ye Tang, Monica Wong-Chan, Joe Zbiciak, and Amin Vahdat. Aquila: A unified, low-latency fabric for datacenter networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1249–1266, Renton, WA, April 2022. USENIX Association.

[21] Google. Snappy, a fast compressor/decompressor. https://github.com/google/snappy.

[22] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct access, High-Performance memory disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 287–294, Carlsbad, CA, July 2022. USENIX Association.

[23] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, Boston, MA, March 2017. USENIX Association.

[24] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. Who Limits the Resource Efficiency of My Datacenter: An Analysis of Alibaba Datacenter Traces. In *Proceedings of the International Symposium on Quality of Service*, IWQoS '19, New York, NY, USA, 2019. Association for Computing Machinery.

[25] Zhiqiang He, Dongyang Wang, Binzhang Fu, Kun Tan, Bei Hua, Zhi-Li Zhang, and Kai Zheng. Masq: Rdma for virtual private cloud. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 1–14,

New York, NY, USA, 2020. Association for Computing Machinery.

[26] Todd Hoff. How Twitter Uses Redis To Scale - 105TB RAM, 39MM QPS, 10,000+ Instances. http://highscalability.com/blog/2014/9/8/how-twitter-uses-redis-to-scale-105tb-ram-39mm-qps-10000-ins.html.

[27] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. An Analysis of Facebook Photo Caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 167–181, New York, NY, USA, 2013. Association for Computing Machinery.

[28] Ying Huang. mm, swap: VMA based swap readahead. https://lwn.net/Articles/716296/.

[29] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 437–450, Denver, CO, June 2016. USENIX Association.

[30] Kartik Kannapur. NYC Taxi Trips - Exploratory Data Analysis. https://www.kaggle.com/kartikkannapur/nyc-taxi-trips-exploratory-data-analysis/notebook.

[31] The kernel development community. BPF Documentation. https://www.kernel.org/doc/html/latest/bpf/index.html.

[32] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. OSv—Optimizing the Operating System for Virtual Machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 61–72, Philadelphia, PA, June 2014. USENIX Association.

[33] Waldemar Kozaczuk. OSv Linux ABI Compatibility. https://github.com/cloudius-systems/osv/wiki/OSv-Linux-ABI-Compatibility.

[34] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. Unikraft: Fast, Specialized Unikernels the Easy Way. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 376–394, New York, NY, USA, 2021. Association for Computing Machinery.

[35] Simon Kuenzer, Anton Ivanov, Filipe Manco, Jose Mendes, Yuri Volchkov, Florian Schmidt, Kenichi Yasukata, Michio Honda, and Felipe Huici. Unikernels Everywhere: The Case for Elastic CDNs. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '17, page 15–29, New York, NY, USA, 2017. Association for Computing Machinery.

[36] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. A Linux in Unikernel Clothing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[37] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a Social Network or a News Media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, page 591–600, New York, NY, USA, 2010. Association for Computing Machinery.

[38] Redis Labs. How fast is Redis? https://redis.io/topics/benchmarks.

[39] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, and et al. Software-Defined Far Memory in Warehouse-Scale Computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 317–330, New York, NY, USA, 2019. Association for Computing Machinery.

[40] Stefan Lankes, Simon Pickartz, and Jens Breitbart. HermitCore: A Unikernel for Extreme Scale Computing. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '16, New York, NY, USA, 2016. Association for Computing Machinery.

[41] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G. Shin. Hydra : Resilient and highly available remote memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 181–198, Santa Clara, CA, February 2022. USENIX Association.

[42] Daan Leijen, Ben Zorn, and Leonardo de Moura. Mimalloc: Free list sharding in action. Technical Report MSR-TR-2019-18, Microsoft, June 2019.

[43] Jiuxing Liu, Wei Huang, Bülent Abali, and Dhabaleswar K. Panda. High Performance VMM-Bypass I/O in Virtual Machines. In Atul Adya and Erich M. Nahum, editors, *Proceedings of the 2006 USENIX Annual Technical Conference, Boston, MA, USA, May 30 - June 3, 2006*, pages 29–42. USENIX, 2006.

[44] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library Operating Systems for the Cloud. *SIGARCH Comput. Archit. News*, 41(1):461–472, March 2013.

[45] Anil Madhavapeddy and David J. Scott. Unikernels: Rise of the Virtual Library Operating System. *Queue*, 11(11):30–44, December 2013.

[46] Dan Magenheimer. Frontswap. https://www.kernel.org/doc/html/latest/vm/frontswap.html.

[47] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is Lighter (and Safer) than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 218–233, New York, NY, USA, 2017. Association for Computing Machinery.

[48] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. ClickOS and the Art of Network Function Virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 459–473, Seattle, WA, April 2014. USENIX Association.

[49] Hasan Al Maruf and Mosharaf Chowdhury. Effectively Prefetching Remote Memory with Leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 843–857. USENIX Association, July 2020.

[50] Hossein Moein. DataFrame. https://github.com/hosseinmoein/DataFrame.

[51] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. A Binary-Compatible Unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2019, page 59–73, New York, NY, USA, 2019. Association for Computing Machinery.

[52] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[53] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, October 2014. USENIX Association.

[54] Jonas Pfefferle, Patrick Stuedi, Animesh Trivedi, Bernard Metzler, Ionnis Koltsidas, and Thomas R. Gross. A Hybrid I/O Virtualization Framework for RDMA-Capable Network Interfaces. *SIGPLAN Not.*, 50(7):17–30, March 2015.

[55] Maksym Planeta, Jan Bierbaum, Leo Sahaya Daphne Antony, Torsten Hoefler, and Hermann Härtig. Migros: Transparent live-migration support for containerised RDMA applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 47–63. USENIX Association, July 2021.

[56] QEMU. Paravirtualized RDMA Device (PVRDMA). https://github.com/qemu/qemu/blob/master/docs/pvrdma.txt.

[57] Ali Raza, Parul Sohal, James Cadden, Jonathan Appavoo, Ulrich Drepper, Richard Jones, Orran Krieger, Renato Mancuso, and Larry Woodman. Unikernels: The Next Stage of Linux's Dominance. In *Proceedings*

of the Workshop on Hot Topics in Operating Systems, HotOS '19, page 7–13, New York, NY, USA, 2019. Association for Computing Machinery.

[58] Linux RDMA. Open Fabrics Enterprise Distribution (OFED) Performance Test. https://github.com/linux-rdma/perftest.

[59] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM. https://github.com/aifm-sys/aifm.

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

[61] Salvatore Sanfilippo. Redis. https://redis.io.

[62] Salvatore Sanfilippo. Simple Dynamic Strings library for C. https://github.com/antirez/sds.

[63] ScaleGrid. Top Redis Use Cases by Core Data Structure Types. https://scalegrid.io/blog/top-redis-use-cases-by-core-data-structure-types/.

[64] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, Carlsbad, CA, October 2018. USENIX Association.

[65] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. Shoal: A Network Architecture for Disaggregated Racks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 255–270, Boston, MA, February 2019. USENIX Association.

[66] Matt Stancliff. Redis Quicklist - From a More Civilized Age. https://matt.sh/redis-quicklist.

[67] Konstantin Taranov, Salvatore Di Girolamo, and Torsten Hoefler. Corm: Compactable remote memory over rdma. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD/PODS '21, page 1811–1824, New York, NY, USA, 2021. Association for Computing Machinery.

[68] Konstantin Taranov, Salvatore Di Girolamo, and Torsten Hoefler. Corm: Compactable remote memory over rdma. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD/PODS '21, page 1811–1824, New York, NY, USA, 2021. Association for Computing Machinery.

[69] Shelby Thomas, Geoffrey M. Voelker, and George Porter. CacheCloud: Towards Speed-of-light Datacenter Communication. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*, Boston, MA, July 2018. USENIX Association.

[70] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: The next Generation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[71] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. Cooperation and security isolation of library OSes for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

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

[73] Chenxi Wang, Haoran Ma, Shi Liu, Yifan Qiao, Jonathan Eyolfson, Christian Navasca, Shan Lu, and Guoqing Harry Xu. MemLiner: Lining up tracing and application for a Far-Memory-Friendly runtime. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 35–53, Carlsbad, CA, July 2022. USENIX Association.

[74] Dan Williams and Ricardo Koller. Unikernel Monitors: Extending Minimalism Outside of the Box. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO, June 2016. USENIX Association.

[75] Wonsup Yoon, Jinyoung Oh, Jisu Ok, Sue Moon, and Youngjin Kwon. Dilos: Adding performance to paging-based memory disaggregation. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '21, page 70–78, New York, NY, USA, 2021. Association for Computing Machinery.

[76] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. XRP: In-Kernel storage functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 375–393, Carlsbad, CA, July 2022. USENIX Association.

[77] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. Carbink: Fault-Tolerant far memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 55–71, Carlsbad, CA, July 2022. USENIX Association.

A Artifact Appendix

A.1 Abstract

This artifact includes DiLOS, its applications, evaluation scripts, and data-sets used for evaluations. Our GitHub repository [4] provides instructions, source codes, scripts, and data-sets for evaluations.

A.2 Description & Requirements

A.2.1 How to access. In our GitHub repository, DiLOS' codes and evaluation workloads are in the `main` branch. Fastswap's codes and workloads are in the `fastswap` branch. In the repository's release section, the data-sets used for workloads are available.

A.2.2 Hardware dependencies. DiLOS requires at least two nodes (one is for a compute node and the other one is for a memory node) which are equipped with Mellanox's ConnectX-5 NICs. The two nodes are directly connected via a 100Gbe link. The nodes should have enough amount of memory/storage to load evaluation workloads (about 40GB of memory and 500GB of storage).

A.2.3 Software dependencies. DiLOS requires Ubuntu 18.04 LTS for evaluation. The artifact provides installation scripts for the other dependencies (kernel, VMM, OFED, etc.).

A.2.4 Benchmarks. All data-sets are available in the release section of our GitHub repository. The artifact also provides downloading and extracting scripts for the data-sets.

A.3 Setup

The artifact provides a README file for setup. We recommend following instructions in the file.

A.4 Evaluation workflow

A.4.1 Major Claims.

- *(C1): DiLOS achieves a higher (compared with Fastswap) or similar (compared with AIFM) performance of state-of-the-art systems for the simple and real-world applications. This is proven by the experiment (E1, E2, E3, E4, E5, E6, E7) described in §6.2.*
- *(C2): DiLOS' app-aware prefetcher achieves 62% higher performance than general-purpose prefetchers for* `LRANGE`*s. This is proven by the experiment (E7) described in §6.3*
- *(C3): DiLOS' app-aware guided paging reduces bandwidth consumption by 12% for DEL on average and 29% for GET operations. This is proven by the experiment (E8) described in §6.3*

A.4.2 Experiments.

Experiment (E1): [Quicksort]: This experiment evaluates DiLOS' performance under quicksort workload (Figure 7(a))
[How to]
Run `./scripts/bench-quicksort.sh` on the compute node.
[Results]
The results will appear in `$HOME/benchmark-out/quicksort/<DATETIME-OF-EVALUATION>` directory.

Experiment (E2): [K-Means]: This experiment evaluates DiLOS' performance under k-means workload (Figure 7(b))
[How to]
Run `./scripts/bench-kmeans.sh` on the compute node.
[Results]
The results will appear in `$HOME/benchmark-out/kmeans/<DATETIME-OF-EVALUATION>` directory.

Experiment (E3): [Compression and Decompression]: This experiment evaluates DiLOS' performance under snappy compression and decompression workload (Figures 7(c) and 7(d))
[How to]
Run `./scripts/bench-snappy.sh` on the compute node.
[Results]
The results will appear in `$HOME/benchmark-out/snappy/<DATETIME-OF-EVALUATION>` directory.

Experiment (E4): [Dataframe]: This experiment evaluates DiLOS' performance under Dataframe workload (Figure 8)
[How to]
Run `./scripts/bench-dataframe.sh`.
[Results]
The results will appear in `$HOME/benchmark-out/dataframe/<DATETIME-OF-EVALUATION>` directory.

Experiment (E5): [GAPBS]: This experiment evaluates DiLOS' performance under GAPBS workload (Figure 9)
[How to]
Run `./scripts/bench-gapbs.sh` on the compute node.
[Results]
The results will appear in `$HOME/benchmark-out/gapbs/<DATETIME-OF-EVALUATION>` directory.

Experiment (E6): [Redis]: This experiment evaluates DiLOS' performance under redis workload (Figure 10)
[How to]
Run `./scripts/bench-redis.sh` on the compute node.
[Results]
The results will appear in `$HOME/benchmark-out/redis/<DATETIME-OF-EVALUATION>` directory.

Experiment (E7): [Redis Bandwidth]: This experiment evaluates DiLOS' bandwidth consumption under Redis workload (Figure 12)
[How to]
Run `./scripts/bench-redis-sg.sh` on the compute node.
[Results]
The results will appear in `$HOME/benchmark-out/redis-sg/<DATETIME-OF-EVALUATION>` directory.

[4] https://github.com/ANLAB-KAIST/dilos