Adios to Busy-Waiting for Microsecond-scale Memory Disaggregation

Wonsup Yoon
KAIST
Daejeon, Republic of Korea
wsyoon@kaist.ac.kr

Jisu Ok
KAIST
Daejeon, Republic of Korea
jisu.ok@kaist.ac.kr

Sue Moon
KAIST
Daejeon, Republic of Korea
sbmoon@kaist.edu

Youngjin Kwon
KAIST
Daejeon, Republic of Korea
yjkwon@kaist.ac.kr

Abstract

How fast and efficiently page faults are handled determines the performance of paging-based memory disaggregation (MD) systems. Recent MD systems employ busy-waiting in page fault handling to avoid costly interrupt handling and context switching. Upon a page fault, they issue a remote fetch request and busy-wait for the completion of the request rather than yield their execution to other tasks. While these attempts succeed to cut the latency of MD systems to microseconds, they suffer from head-of-line (HOL) blocking that leads to high tail latency and causes RDMA network underutilization.

To address the problems, we reload the yield-based mechanism into the page fault handling and propose a new MD system, Adios. While yielding involves the switching overhead, Adios minimizes it by putting the page fault handler and the execution scheduler into a single address space. Then we use newly designed lightweight user-level threads, namely unithread. We also devise a dispatching algorithm that alleviates the imbalance in RDMA queue pairs, assuring lessened queueing delays and improved RDMA network utilization. Our evaluation demonstrates that Adios outperforms an existing state-of-the-art busy-waiting MD system, DiLOS, by up to 1.07-1.64× in throughput and 1.99-10.89× in P99.9 latency on real-world applications.

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

Keywords: memory disaggregation, disaggregated data center, unikernel, microsecond-scale

ACM Reference Format:
Wonsup Yoon, Jisu Ok, Sue Moon, and Youngjin Kwon. 2025. Adios to Busy-Waiting for Microsecond-scale Memory Disaggregation. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3689031.3717475

1 Introduction

Memory disaggregation (MD) systems decompose a computing system into separately managed compute and memory nodes connected by a fast interconnect such as RDMA. By dynamically allocating memory resources on demand, they improve memory utilization and memory scaling across hardware boundaries. They also gain availability from separate hardware fault domains [10].

Diverse approaches to MD systems have emerged over time. Those in the exploration of application semantics have implemented MD in libraries or runtimes. Others have chosen transparency and opted for implementation in an OS kernel. Kernel-based MD systems extend paging-based memory systems and support existing applications to run without modification [4, 21, 44, 48, 54, 60, 64]. Infiniswap, the first MD system based on paging [21], and later paging-based systems use RDMA-connected remote memory as a backing store.

MD systems with busy-waiting. At the core of paging-based systems lies page fault handling. When an application accesses memory uncached in local DRAM, a page fault occurs, and the page fault handler fetches the faulted page from a remote memory node. While the NIC fetches pages in background, traditional page fault handlers *yield* their contexts for CPU to execute other tasks. Once the page fetch is complete, interrupt handling and context switching take place, resulting in more than 100−1,000 μsec of latency [21]. In order to overcome this latency penalty, recent MD systems have adopted *busy-waiting* in page fault handling [4, 44, 48, 64]. The rationale behind busy-waiting is that remote memory access via RDMA is faster than interrupt handling and scheduling. Fastswap has adopted busy-waiting and reported a 3× throughput improvement over Infiniswap with yield-based page fault handling [4].

The rise of low-latency I/O devices has changed the time scale of event handling in modern datacenters; they have pushed systems to review existing layering and abstractions and to optimize in the microsecond scale [7, 15, 39]. Recent MD systems, such as Hermit and DiLOS, push the performance into the microsecond scale by building efficient code paths for busy-waiting page fault handlers. Hermit has removed non-urgent operations in the page fault handler and processes them asynchronously. It "reduces the 99th percentile tail latency by 99.7%" in Memcached against Fastswap [48]. DiLOS builds upon the OSv unikernel, and its lightweight page fault handler cuts down the Redis GET latency to just a third of Fastswap's. It consolidates all paging-related data structures into a unified page table in order to perform page fault handling with a single lookup.

Shortcomings of busy-waiting. With the advent of RDMA in modern computing servers, busy-waiting has become a viable solution. Yet, it has shortcomings. First, tasks queued behind a busy-waiting task experience head-of-line (HOL) blocking. HOL blocking exacerbates when queues are partitioned or the workload has high dispersion [28, 47]. One more issue with busy-waiting is limited concurrency. Modern NICs allow multiple outstanding requests. If the tasks are busy-waiting, the maximum number of outstanding requests is small and suboptimal. This results in NIC and networking underutilization, limiting potential gain in throughput.

Lightweight user-level threading. The above shortcomings of busy-waiting have made us revisit its original motivation: avoiding interrupt handling and context switching overhead. The source of context switching overhead is heavy-weight kernel threads and their schedulers [5]. In contrast to kernel threads and schedulers, a user-level scheduler creates multiple stacks within a process and switches contexts by manipulating the stack pointers. Since all operations in user-level context switching are unprivileged, neither expensive mode switching nor kernel operations occur, enabling sub-microsecond context switching. Thanks to its lightness, user-level threading has been widely adopted for microsecond-scale scheduling [26, 28, 46, 49, 58]. Capriccio is an early effort to utilize user-level threading to support tens of thousands of concurrent requests [58]. Arachne's lightweight context switching allows user-level threads to adapt to a dynamically changing number of CPU cores [49]. Shenango enhances its core allocator with busy-spinning and succeeds in cutting down the CPU core reallocation time to as low as 5 μs [46]. Shinjuku, on the other hand, extends user-level threads to support preemption at the microsecond scale [28]. It reduces the overhead of preemptive scheduling by directly accessing IPI (Inter Processor Interrupt) hardware using virtualization technology in its single address space and protection domain. Given a high dispersion workload, Shinjuku approximates the PS (Processor Sharing) scheduling and drops the tail latency from milliseconds to microseconds. Concord enhances the preemptive scheduler using compiler instrumentation rather than IPI [26]. It inserts yield points during compilation, and applications voluntarily yield their contexts if requested at such points. By shifting to compiler-based preemption, Concord cuts down the preemption overhead to 1/4 of Shinjuku's.

Limitations of user-level threading. For all the advantages that user-level threads and schedulers offer, it is not straightforward to adopt them in MD systems. Since page faults are kernel-level events, user-level schedulers have no direct way to be notified. The page fault handler has to wake up a user-level handler [5] and the scheduler follows up. In Linux, `userfaultfd` is widely used to handle page fault events at user-level [2]. It was originally developed to support VM live migration [1], but later also applied to other uses. Canvas uses it to design two-tier prefetching for MD [61]. When the default kernel prefetcher is not effective, Canvas forwards prefetching events to the application tier. The application computes pages based on the user-level thread's access patterns and prefetches them. zIO designs transparent zero-copy I/O by tracing application accesses to buffer via `userfaultfd` [55]. Its main insight is that the application touches only a small portion of the I/O buffer during transmission, and it copies only the touched areas. User-level handlers, such as `userfaultfd`, still add a few microseconds to the overall request service latency [63].

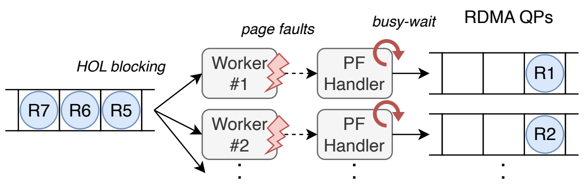
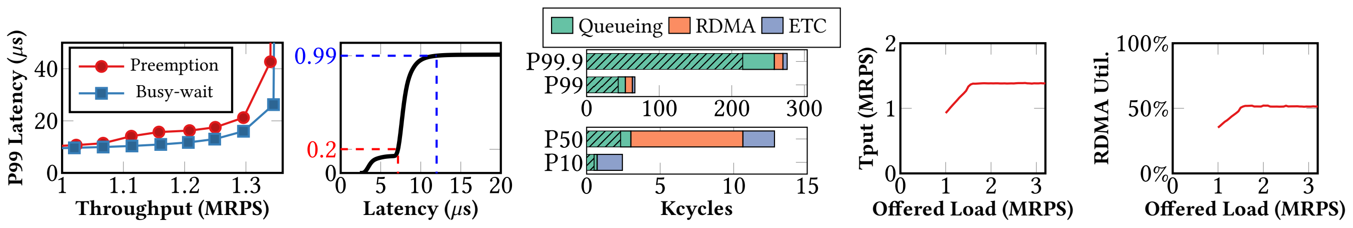


Figure 1. Architecture of a single queue request handler. Requests flow from left to right. If any page fault, page fault handlers issue RDMA commands and busy-wait.

Sketch of Adios design. In order to take advantage of light user-level threading and to achieve microsecond latency in MD systems, we presume that the page fault handler and the scheduler should reside in the same address space and protection domain. This has led us to design a new memory disaggregation system, Adios, based on a single address space. (1) In this work, we choose to use a unikernel for its compact code base. In our system, the page fault handler yields directly to the scheduler upon issuing a page fetch and returns to the thread upon page fetch completion. This direct switching between the handler, the scheduler, and threads is efficient. The scheduler implements the single queuing policy, which has proven to achieve the best tail latency [26]. (2) Our threads should be as light as user-level threads but carry out kernel functions such as page fault handling. Under a heavy workload of millions of requests per second, a thread per request may build up to a significant portion of local memory and limit the cache size for remote memory. We have designed and implemented a unithread that occupies a minimal memory footprint but uses unified data structures for both kernel and user stacks. (3) The scheduler employs a dispatching algorithm that selects, among idle workers, one with the least number of in-flight page fetches. The dispatching algorithm alleviates imbalances in RDMA QPs (Queue Pairs).



(a) P99 latency **(b)** CDF of latency **(c)** Latency breakdown **(d)** Request throughput **(e)** RDMA utilization

Figure 2. Performance analysis of DiLOS [64]. For (b) and (c), the throughput is 1.3 MRPS. In (c), queueing delay due to busy-waiting is marked with slashes.

Evaluation and conclusion. For evaluation, we compare Adios with busy-waiting page fault handling systems, Hermit [48] and DiLOS [64]. In our evaluation, we also include a modified version of DiLOS with a preemptive scheduler similar to Concord [26]. This scheduler preempts long-running tasks and executes other tasks during page fetching. It is a representative user-level scheduler design that addresses the HOL blocking problem. Yet without cooperation from the page fault handler, the preemptive scheduler is oblivious to busy-waiting and does not discern such tasks. In microbenchmark evaluation, we demonstrate that Adios' yield-based handler succeeds in eliminating HOL blocking, resulting in a precipitous drop in tail latency and a 58% increase in RDMA link utilization. We use four real-world applications, Memcached [14], RocksDB [24], Silo [57], and Faiss [17], to demonstrate that Adios carries MD into the realm of the microsecond scale. Overall, Adios outperforms Hermit and DiLOS by achieving up to 1.67-4.97× and 1.07-1.64× higher throughput; and 1.83-4.30× and 1.99-10.89× better P99.9 latency, respectively. Compared to DiLOS with a preemptive scheduler, Adios delivers 2.71× better P99.9 GET latency under a RocksDB workload mixed with 99% GET/1% SCAN (100), where preemptive scheduling is effective.

Below, we summarize our contributions:

- We present a novel MD system, Adios, that is based on yield-based page fault handling. To the best of our knowledge, Adios is the first MD system to achieve a single-digit microsecond-scale tail latency.
- We show that Adios eliminates HOL blocking problem cases from busy-waiting with our yield-based page fault handling and page-fault-aware dispatching algorithm.
- Our Adios delivers greatly improved performance over existing systems in real-world workloads: tail latency (1.99-10.89×) and throughput (up to 1.07-1.64×).

Source code. The source code of Adios is available at https://github.com/ANLAB-KAIST/adios.

2 Motivation

Before we present our Adios design, we delve into the inefficiencies stemming from busy-waiting in MD systems. Microbenchmark analysis of these inefficiencies has been the motivating guideline for us to design our MD system.

For our investigation, we choose DiLOS [64], one of the most recent MD systems with the best-reported performance and publicly available source code. On top of DiLOS, we implement a request handler designed to process networked requests from clients. The architecture follows a single queue model, akin to other well-known microsecond-scale systems [16, 26, 28]. Figure 1 illustrates the system configuration. When networked requests arrive, they are distributed from a central queue to the workers. In case of a page fault in a worker, the page fault handler takes control and initiates a retrieval of the remote page by issuing an RDMA request. The page fault handler busy-waits for the completion of the outstanding RDMA fetch request.

For our analysis, we set the local cache size to 20% of the total working set and vary the offered load. We measure the latency and analyze its contributing factors. From the throughput measurement, we analyze whether or not the throughput matches the RDMA bandwidth at full utilization.

We use an open-loop load generator to emulate a large number of clients. It runs on a node separate from our compute and memory nodes. Additional details about the load generator are in §4.

2.1 Tail Latency Breakdown

We measure the latency under a heavy workload and break down the main contributing factors in the tail latency. We configure the request handler to have an array whose size does not fit in the local memory; only 20% of the working set fits in the local memory, and the rest is in remote memory.. The handler receives requests, each of which contains a random index of the array. The handler replies with the value at the index.

We first measure the end-to-end (e2e) latency, which is calculated by subtracting the TX timestamp from the RX timestamp at the load generator. This end-to-end latency includes the network delay between the load generator and all delays (*e.g.*, queueing, processing, and remote memory) in computing nodes. We plot the 99th percentile[1] of the e2e latency against the varying workload along the x-axis in Figure 2(a). At around 1,300,000 requests per second (1.3 MRPS), the P99 latency starts to increase rapidly (labeled 'Busy-wait'

[1]We use the notation of Px for the x-th percentile for the rest of this paper.

in Figure 2(a)). We leave the graph labeled 'Preemption' for later discussion in §2.3. To investigate further, we plot the latency distribution of DiLOS at 1.3 MRPS and its request handling breakdown in Figures 2(b) and 2(c), respectively. The cumulative distribution function (CDF) of the latency outlines three distinct regions of performance: below P20 (the dotted red line), below P99 (the dotted blue line), and above P99. The requests with latency below P20 are those served from local memory, as we have configured a 20% local memory ratio for the working set. Beyond P99, we observe that the latency is about 10× higher than the latency below P20. Figure 2(c) plots the breakdown of the request handling latency at P10, P50, P99, and P99.9. The x-axis is in cycles, measured using `rdtsc()` on the compute node. Note that this latency does not cover the connection between the load generator and the compute node; only from the point a request arrives at the compute node and till its reply departs. At P10, there is no RDMA overhead in the latency, while at P50, the RDMA overhead takes up more than half of the overall cycles. Here, the range of the x-axis is up to 15 Kcycles. From P50 to P99, the overall latency jumps by 5×, and from P99 to P99.9, it jumps more than 5×. Pay attention that the x-axis now plots up to 300 Kcycles. From P50 to P99.9, there is more than an order of magnitude difference, most of which comes from the queueing delay.

In Figure 2(c), we have marked with slashes among queueing delay the time spent in busy-waiting. They are negligible at P10 and P50 but become dominant in P99 and P99.9. From this analysis, we conclude that HOL blocking is the dominant factor in high queueing delays.

2.2 Under-utilized RDMA

Next, we examine the impact of busy-waiting on the throughput. We use the same configuration as the previous experiment but measure the request handling throughput under varying loads.

Figure 2(d) shows the measurement result. As we vary the load along the x-axis starting from 1 MRPS up to 3 MRPS in the unit of 50 KRPS, at around 1.4 MRPS, the request handling throughput stalls. The gap between the offered load and the throughput translates to dropped requests. We observe that as the throughput approaches 1.4 MRPS, the system is saturated: the P99 latency shoots up as seen in Figure 2(a), and the throughput stagnates at around 1.38 MRPS as seen in Figure 2(d).

Is the throughput capped by busy-waiting or by the RDMA bandwidth? Here, we check to see if the RDMA network bandwidth is saturated. Figure 2(e) shows the RDMA link utilization from the experiment. Even in the range of offered load above 1.4 MRPS, where the system drops requests, the RDMA link shows utilization of only 50% and has room for more traffic.

2.3 Limitations of Existing Approaches

So far, we have seen in detail the impact of busy-waiting in page fault handling. Now, how should we address the shortcomings of busy-waiting? First, let us look at existing approaches.

Overlapping computation with I/O. To reduce wasted cycles from busy-waiting, MD systems overlap computation with page fault I/O. Executing a prefetching algorithm is one of the most common tasks chosen for overlapping [4, 44, 48, 64]. After issuing an I/O request but before starting busy-waiting, these systems compute prefetching algorithms and issue additional page fetches. On top of prefetching, Hermit also overlaps metadata updates and cgroup accounting with I/O. However, processing these tasks accounts for only 10% of the entire page fetching latency, and the remaining 90% of the cycles are still wasted [48].

User-level threading. As pointed out in other works [4, 44, 48, 64], a general-purpose kernel scheduler is too heavy to support context switching during fast I/O. To reduce the high context switching overhead, many systems adopt user-level threads and schedulers [26, 28, 46, 49, 58].

Preemptive scheduling mitigates HOL blocking in scenarios where a long-running task (busy-waiting in our case) blocks short-running tasks. Shinjuku [28] and Concord [26] have demonstrated that preemption is viable for microsecond-scale scheduling. To check the effectiveness of preemption in user-level threading, we conduct the same experiment as in §2.1 but with a preemptive scheduler added to DiLOS. It closely follows the design of Concord. We defer a detailed description of the preemptive scheduler implementation to §5. The scheduler uses the 5 μs preemption interval, which is the default value of Shinjuku and the minimum value without significant system overhead in Concord. We plot the results in a graph labeled 'Preemption' in Figure 2(a). Just from the 5 μs preemption interval and a typical 2-3 μs delay for a 4KB page fetch over RDMA, we observe that the preemptive scheduling does not improve the performance of MD systems but rather deteriorates the P99 latency. Preemptive scheduling alone in user-level threading is of little impact.

Limitations of user-level threading. From the microbenchmark analysis in this section, we have seen that current busy-waiting page fault handling incurs much queueing, especially at the tail of the latency distribution, and leaves much room for improvement in network utilization. As we have argued in §1, the main hurdle for adopting user-level threading in MD systems is that the page fault handler and the user-level scheduler reside in separate protection domains and are oblivious to each other. Next, we list the challenges for a new MD system.

2.4 Our Approach and Challenges

We begin the design of a new MD system departing from today's busy-waiting MD systems. Upon issuing an RDMA page fetch command, the page fault handler *pauses* and switches contexts swiftly to other user-level threads. In order to build such a page fault handler in the same protection domain as the scheduler and threads, we need to address the following challenges.

Challenge #1: Limited features of user-level threads. User-level threads lack features that kernel-level threads have, such as exception handling. In order for a user-level thread to deal with exception handling, it needs its own exception stack. It also should handle page table walking, prefetching, and I/O of its own exceptions. This exception handling increases both the computation and memory overheads of threading. We need an efficient mechanism to curb the overhead.

Challenge #2: High cooperation cost. Yield-based page fault handling requires close cooperation with the user-level scheduler, the kernel page fault handler, and the I/O devices. Having them in separate protection domains costs highly at boundary-crossing. Focusing on a single component or layer alone has limited impact.

Challenge #3: Uneven distribution of page fault requests. With yield-based fault handling, more RDMA commands are issued at any given time, and they result in increased RDMA throughput. Some workers may have an unfair share of page faults, because not all requests cause page faults. Some RDMA QPs, in turn, may have more outstanding requests than others. This unfair distribution in RDMA requests generates a transient long queueing delay in some QPs and pushes the tail latency to go up.

3 Design of Adios

As discussed in §2.4, reaping the performance benefits of yield-based page fault handling at the microsecond-scale necessitates revamping the entire datapath of memory disaggregation and a new abstraction for lightweight context switch. Without such reconsideration, the major overhead arises from context-switching and communication costs between the kernel and the scheduler, negating the advantages of yield-based page fault handling. In response, we do not limit our effort to a single layer but across all layers–the kernel, the scheduler, and the application–to fully harness the advantages of yield-based page fault handling.

Address Challenge #1: Formulate a new abstraction for lightweight threads. The page fetch latency in modern 100GbE RDMA NICs is as low as 2-3 microseconds [29, 64, 66]. This latency draws an upper bound for the context-switching time from the yield-based page fault handler to an application thread. To meet the strict latency requirement, we introduce *unithread*, a lightweight abstraction that provides fast context switching and kernel features to perform page fault handling. Moreover, the unithread employs *universal stack design* that consolidates all stack data for user and kernel-level threads into a single stack, reducing memory fragmentation across kernel and user stacks.

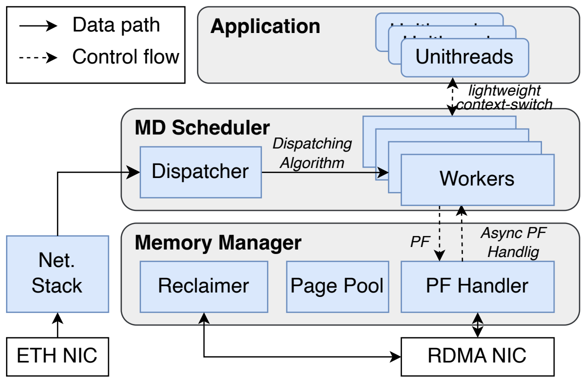


Figure 3. Adios' computing node architecture

Address Challenge #2: Adopt unikernel to run all components in a single address space and protection domain. To build the yield-based page fault handling mechanism, we implement the Adios kernel as a unikernel. In a conventional architecture, applications and their libraries operate within the user space, while the page fault handler resides in the kernel space. The control transfer time in Intel x86 using the `iret` instruction takes 1-2 μs. Furthermore, the kernel cannot directly transition to user-level contexts, as mechanisms such as Intel SMEP (Supervisor Mode Execution Prevention) or ARM PXN (Privileged Execute-Never) strictly forbid direct control jumps from the kernel to the user level. In a unikernel instance, there is no longer a distinction between user and kernel components, as all components are in the same address space and protection domain, eliminating any boundary-crossing cost.

Address Challenge #3: Devise a fair mechanism to break the uneven page fetches over RDMA QPs. Faced with uneven distribution of page fetches across RDMA QPs, we should allocate a worker to an arriving task in such a way to break the unevenness, if any. We devise a task-dispatching algorithm that takes into account the number of outstanding page fetches per RDMA QP.

In this section, we first present an overall architecture of Adios in §3.1 and the unithread, core data structure of Adios in §3.2. Adios consists of two main parts: the memory manager performing yield-based page fault handling and the MD scheduler executing unithreads. We describe the memory manager in §3.3 and the MD scheduler in §3.4.

3.1 Adios Architecture

Figure 3 illustrates the overall architecture of Adios' computing node. The memory manager is responsible for providing remote memory functionalities to the scheduler, including page fault handling and reclamation. Similar to previous MD systems [4, 21, 44, 59, 60, 64], the computing node leverages one-sided RDMA due to its performance benefits over two-sided RDMA. The MD scheduler consists of a dispatcher

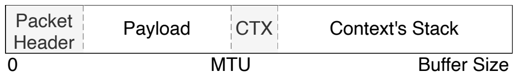


Figure 4. Unithread's memory layout.

and workers. The dispatcher distributes networked requests from clients across the workers, and the workers process the requests.

Application execution model. To handle multiple requests concurrently, Adios follows a thread-based architecture similar to prior studies [26, 28]. Upon receiving a new request, a worker creates a new unithread dedicated for the request and context-switches to the unithread. In the unithread, the application processes the request and context-switches back to the worker at the end.

3.2 Unithread

The goal of the unithread is to provide both kernel and user features while consuming minimal memory footprint. Adios pre-allocates memory for unithreads to avoid runtime allocation overhead, and the number of pre-allocated unithreads must be sufficient to handle bursty request arrivals. We have to meet conflicting goals of pre-allocating sufficient memory for unithreads but minimizing the overall memory footprint. The number of pre-allocated unithreads is configured and fixed to 131,072 in total, but it can be lowered to reduce memory usage or increased to tolerate workloads with more intensive bursts of requests. We focus our memory-saving effort on compact data structures for unithreads.

Data structure. To reduce the memory used for threading, each unithread is designed to have a memory footprint as small as possible. In particular, Adios allocates a single buffer that contains both a network request and a unithread's context to execute the request (Figure 4). On receiving a network request, the networking stack stores the packet payload at the head of the buffer. At the end of the packet payload, unithread context data structure and the stack area follow. The context data structure contains just enough information to save and restore context necessary for executing application's code, minimizing the memory footprint and thus context-switch time. That is, a unithread context only includes one argument register and five callee-saved registers (`rbp`, `rip`, `rsp`, `mxcsr`, and `fpucw`). The rest of the registers, including floating pointer registers, are stored in the caller's stack frame if necessary [42]; hence, there is no need to save and restore them. Without dumping the entire floating point registers, unithread achieves low overhead context-switching. Compared to Shinjuku's user-level threads, the unithread is far faster in context switching. We run microbenchmarks to compare the cycles in context switching. Our unithread context switching is 4.7× faster and uses 12.1× smaller memory than Shinjuku's `ucontext_t`, which provides equivalent functionality (Table 1).

Mechanism	Context Size	Cycles
Adios' unithread	80B	40
Shinjuku's `ucontext_t`	968B	191

Table 1. Comparison of context-switching mechanisms

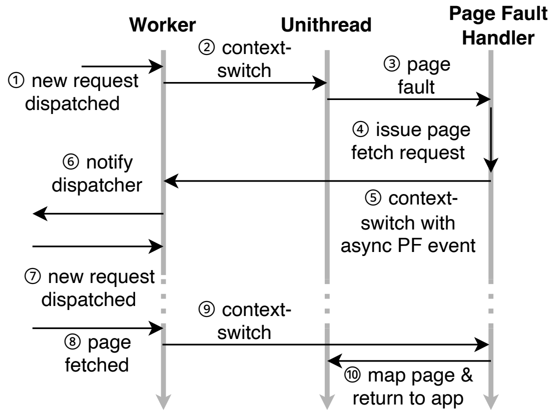


Figure 5. Yield-based page fault handling procedure

Universal stack. To further reduce the memory footprint of the unithreads, Adios employs a single shared stack, *universal stack*. In conventional OSes, the user-level and the kernel code operate on separate stacks to isolate kernel data from user-level code. Adios merges the kernel and application stacks into one, allowing each unithread to use a universal stack for both the kernel and the user code. This universal stack design reduces internal fragmentation compared to the traditional two-stack model, eventually reducing memory usage. It is worth noting that this co-location is possible because we co-design the unikernel and the scheduler.

Overall, a unithread consumes a minimum of 4KB[2] of memory per request, whereas in Shinjuku, each request needs at least 12KB: 4KB each for packet payload and context, user-level stack, and exception (kernel) stack. As a result, a unithread has a 66% smaller memory footprint per request, and in turn, it reduces 66% of pre-allocated buffers (131,072 in total), freeing 1GB (which corresponds to 12.5% of the 8GB local DRAM cache size used in the experiments).

3.3 Yield-based Page Fault Handling

In the yield-based page fault handling, the page fault handler context-switches back to the worker instead of waiting for the page fetch to complete, thereby eliminating busy-waiting in the code path. Figure 5 describes the procedure of our yield-based page fault handling. ① A worker receives a new request from the dispatcher and ② starts handling the request by context-switching to its application unithread. ③ If the application unithread accesses a page that does not reside in the computing node's local memory, a page fault exception occurs. Then, ④ the page fault handler issues a yield-based page fetch request to the memory node and ⑤ yields its

[2]The stack size may grow if an application requires larger stack.

context to the worker. ⑥ The worker informs the dispatcher to receive and ⑦ handles a new request. Before starting new unithreads, the worker polls the NIC's queue once to check page fetch request completions. Later, ⑧ the outstanding page fetch request completes, the request is polled, and ⑨ the worker context-switches back to the page fault handler. Lastly, ⑩ the page fault handler maps the fetched page and returns to the original request handler.

Reclaimer. If no local cache page is available, the page fault handler is unable to allocate a new page and must pause until the reclaimer evicts a page. In many OSes such as DiLOS, reclaimer threads wake up on high memory pressure and start evicting unused pages. Since the threads are managed by schedulers, the wake up of the reclaimer threads is subject to scheduling overheads as well as delays due to execution of other tasks. If the wake up process takes too long, page allocation overtakes page reclamation, resulting in an out-of-memory state that freezes the whole system. In contrast, Adios employs a pinned dedicated thread that reclaims pages proactively to eliminate the wake up process. Our proactive page reclamation works as follows. The reclaimer thread monitors the current level[3] of memory use and proactively evicts pages before entering an out-of-memory state. The reclaim process is highly responsive because the thread is pinned to a CPU core and starts reclamation immediately after reaching a certain threshold of memory pressure.

3.4 MD Scheduler

The MD scheduler distributes requests and unithreads across CPU cores. Its primary performance objective is to minimize the request handling latency. To meet this objective, Adios employs several scheduling techniques, including single queueing, PF(page fault)-aware dispatching, and polling delegation.

Single queueing. Adios' scheduling policy is based on single queueing. One or a few centralized dispatchers receive requests from the network, and they assign the requests to available workers. This mechanism alleviates HOL blocking and cuts down the tail latency. Many systems employ this design [16, 26, 28].

Single queueing follows a centralized FCFS (First-Come First-Serve) policy and does not perform work-stealing (approximated centralized FCFS) [47]. Both centralized and approximated centralized FCFS queuing policies reduce load imbalance across workers. Yet the work-stealing algorithm needs to scan the queues, which is sub-optimal for low dispersion and highly concurrent workloads [28], and thus we do not adopt it in our system design.

PF-aware dispatching for temporary page fault imbalance. In a system with a busy-waiting page fault handler, each worker issues only one outstanding request in its RDMA QP. In contrast, in Adios, each worker makes multiple outstanding RDMA requests, allowing for greater concurrency but also introducing the risk of load imbalance across RDMA QPs. Even when the dispatcher evenly distributes requests to workers in a round-robin manner [26, 28], the number of RDMA requests can become uneven because not all requests trigger page faults. Consequently, certain workers end up temporarily handling a higher volume of page faults than others. This burst of page faults experienced by a specific worker leads to longer RDMA queue lengths than their peers, leading to an increased RDMA latency due to queueing delay.

Algorithm 1 PF-aware dispatching

```
procedure DispatchReqs(workers, pendingRequests)
    $ready \leftarrow EmptyList()$
    for all $worker \in workers$ do
        if $worker.isIdle()$ then
            $ready.pushBack(worker)$
    $dispatchOrder \leftarrow SortByOutstandingPFCount(ready)$
    while $pendingRequests.isNotEmpty()$ do
        if $dispatchOrder.isEmpty()$ then return
        $req \leftarrow pendingRequests.popFront()$
        $worker \leftarrow dispatchOrder.popFront()$
        $DispatchRequest(worker, req)$
```

To mitigate the temporary page fault imbalance problem, one can use load imbalance-mitigation policies, such as single queueing and work-stealing. However, the single queue policy for RDMA QPs degrades RDMA performance since it limits NIC parallelism [29] and requires costly inter-thread locking to access shared queue. Likewise, work-stealing is infeasible because the RDMA QPs are hardware resources handled by the NIC, which does not support work-stealing.

Therefore, we devise page-fault-aware (PF-aware) dispatching to resolve the temporary imbalance problem at the dispatcher level (Algorithm 1). The core idea behind the algorithm borrows the concept of congestion from network load balancing. In congestion-aware load balancing algorithms, packet steering is decided based on congestion signals [3, 30, 31, 50, 62, 65]. Similarly, Adios uses the RDMA QP's length as an indicator of congestion and prioritizes less congested workers. That is, the dispatcher selects workers that have fewer outstanding page faults than others first and dispatches requests to them. The dispatching algorithm mitigates the concentration of page faults to certain workers and alleviates the long tail latency due to the queueing delay by temporary page fault imbalance. To implement this dispatching algorithm, the scheduler has to access the information of RDMA QPs used by the page fault handler, which is in the kernel device driver. Adios implements this dispatching algorithm without having huge overhead because the user-level scheduler directly accesses the kernel-level QP information exposed by the unikernel.

[3] 15% of total local memory by default, but can be adjusted.

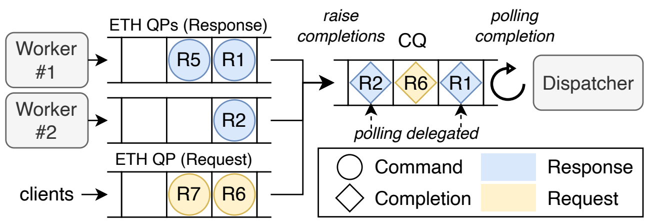


Figure 6. Polling delegation mechanism

Polling delegation. To improve not just the page-fault handling but also the end-to-end performance of networked requests, Adios attends to the exit path of a request reply. During a reply packet transmission, a worker should wait for the completion of the transmission. Once the transmission is complete, the worker informs the dispatcher to recycle the packet buffer for subsequent requests. However, a naive design of the transmission waiting mechanism introduces another HOL blocking problem. If a worker busy-waits for a response transmission completion, it blocks the opportunity to process other requests from the dispatcher, leading to similar problems caused by busy-waiting page fault handling (§2.1).

In order to prevent the HOL blocking, each worker delegates completion polling to the dispatcher. In the NIC's CQ (Completion Queue) and QP semantic, a CQ can manage multiple QPs, and when an I/O event completes on a QP, its completion event is raised to the associated CQ. Adios leverages this semantic to redirect I/O completions to the dispatcher. Figure 6 shows the polling delegation mechanism. When a worker replies to clients, it issues a reply command to its own QP (blue bubbles). After its transmission completes, Adios makes the completion notification raised in the dispatcher's CQ (blue diamonds) instead of using the worker's CQ, and the dispatcher receives the completion notification via polling and recycles the buffer used by the request. Using this mechanism, workers avoid busy-polling in the response path. This mechanism introduces negligible polling overheads to the dispatcher because the dispatcher is already polling the CQ for incoming packets. Furthermore, the dispatcher can batch handling of requests and response completions.

4 Implementation

We have implemented Adios on the OSv unikernel version 0.55 [34]. Adios' core is written in 8,396 LoC[4] of C, C++, and assembly languages. The source code of Adios is available at https://github.com/ANLAB-KAIST/adios.

RDMA and Ethernet networking. Adios' computing node utilizes both RDMA and user-space Ethernet networking on top of unikernel. To use RDMA at the unikernel, Adios reuses DiLOS's libibverbs library, which employs the VMM-bypass mechanism [41]. It maps the NIC's MMIO and DMA regions to the VM's physical memory, enabling direct access to the NIC inside the VM. To support the user-space Ethernet networking, Adios uses NVIDIA OFED's Raw Ethernet feature [12]. Compared with the other popular kernel-bypassing solutions [19, 33, 52], it has several advantages. First, since the Raw Ethernet is a feature of libibverbs library, we can port the feature to unikernel with a tiny modification on DiLOS's RDMA library. Second, it shares the same data structures with the RDMA stack (*e.g.*, `ibv_mr`, `ibv_pd`, `ibv_cq`, etc.). For systems like Adios, which use both RDMA and Ethernet, sharing the data structures simplifies implementation and code path. Lastly, the Raw Ethernet feature's CQ/QP semantic is well aligned to the polling-delegation mechanism, as we discussed in §3.4.

Load generator. To emulate a workload from a large number of clients, we have implemented an open-loop load generator similar to mutilate [38] and used a Poisson process. The load generator also employs the Raw Ethernet feature to send and receive packets, bypassing the kernel for low latency. Apart from kernel-bypassing, the Raw Ethernet feature allows efficient reads of the NIC's hardware timestamps [6]. When the feature is turned on, the NIC records the timestamps of TX and RX at completion descriptors, and the load generator computes the request-response latency by subtracting the TX timestamp from the RX one.

5 Evaluation

In this section, we evaluate the performance of Adios using microbenchmarks and real-world application workloads.

Testbed. We setup a computing node, a memory node, and a load generator emulating clients. Two 100GbE Ethernet cables connect the computing node with the memory node and the load generator. Both the computing and memory nodes have an Intel Xeon Gold 6330 CPU 2.00GHz, 256 GB DDR4 3200 MHz memory, and an NVIDIA ConnectX-6 Dx 100GbE card (CX623106A) each. The load generator has an Intel Xeon Gold 6226R 2.90GHz, 384 GB DDR4 2933MHz memory, and an NVIDIA ConnectX-5 EDR + 100GbE card (CX556A). The computing node runs on top of QEMU 4.2.1 and Ubuntu 20.04 with Hermit [48]'s modified Linux kernel 5.14. For all experiments, the load generator and the memory node use Ubuntu 20.04 with Linux kernel 5.4. NVIDIA OFED 5.8 is used for all three nodes.

Setup. We configure all experiments to use eight worker threads, one dispatcher thread, and one reclaimer thread. We compare Adios with three baseline systems: Hermit (`Hermit`), busy-waiting page fault handling (`DiLOS`), and busy-waiting page fault handling with preemption (`DiLOS-P`). Hermit [48] is a state-of-the-art kernel-based MD system, which adopts asynchronous design in page fault handling but yet still relies on busy-waiting as discussed in §2.3. The busy-waiting page fault handling system is based on the original implementation of DiLOS [64], which employs busy-waiting to check

[4]We use SLOCCount to measure LoCs.

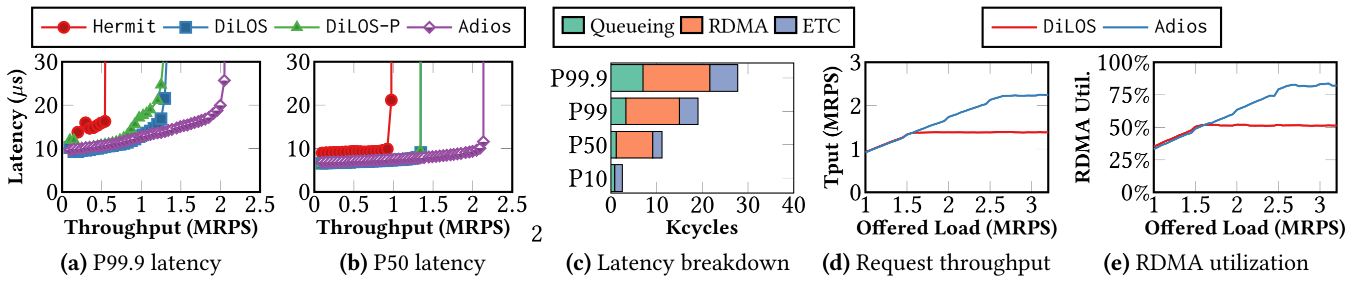


Figure 7. Performance analysis of Hermit, DiLOS, DiLOS-P, and Adios. (a) shows P99.9 latency and (b) P50. (c) shows performance breakdown of Adios's request handling where the throughput is 1.3 MRPS. (d) and (e) show throughput and network utilization of Adios and DiLOS.

the completion of page fetches. We also extend DiLOS to perform preemptive scheduling. Shinjuku [28] and Concord [26] have demonstrated the advantages of preemptive scheduling in the reign of microsecond-scale tail latency. Their preemptive scheduling is based on inter-processor-interrupt (IPI) or compiler-enforced cooperation with the default 5 μs preemption interval. We have tested DiLOS-P with the same 5 μs interval and with both IPI and manually enforced cooperation[5]. The latter required minor code modification but has superior performance than the former with IPI. Thus, we use DiLOS-P with manually enforced cooperation in our evaluation. To validate our implementation of DiLOS-P, we have run the same sets of "Shinjuku-SQ without Preemption" and "Shinjuku-SQ" on RocksDB from [28] and seen comparable results. We have also considered other scheduler designs [16, 46, 47] but dropped them from our evaluation because they employ the run-to-completion model similar to DiLOS. For fair comparisons, all the systems under testing use 2MB huge pages for memory nodes and 4KB pages for compute nodes. Since the original Hermit's memory server implementation uses 4KB pages only, we modified the memory server to employ 2MB huge pages for remote memory. We also considered Infiniswap [21] for evaluations. However, it shows very high P99.9 latency (582 μs to 73 ms) and low throughput (261 KRPS), which are hard to include in figures of relevant scales. Therefore, we exclude the results for space and focus on comparisons between Hermit, DiLOS, and Adios.

5.1 Microbenchmark

We begin our evaluation with microbenchmarks, the same random index indirection workload, as in §2, now with all four systems. Clients send requests with a random index of an array, and the computing node reads and responds with a value at the index. The size of the array is 40GB and the computing node has 8GB local cache, which is 20% of the total working set.

Latency. Figures 7(a) and 7(b) show the P99.9 and median latency of Hermit, DiLOS, DiLOS-P, and Adios over a range

[5]Humans rather than compiler insert probes for checking preemption.

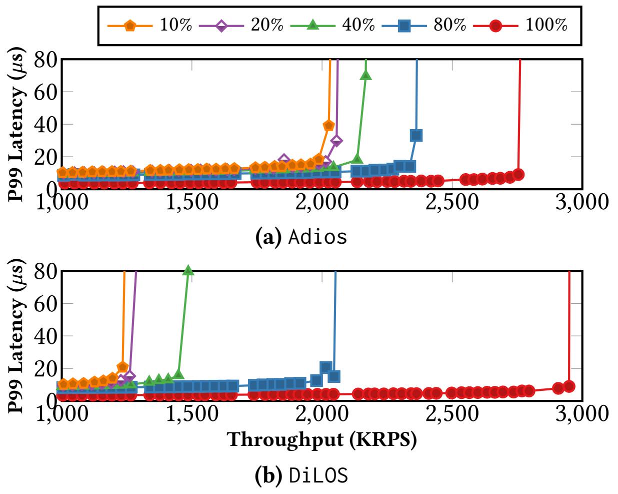


Figure 8. P99 latency of DiLOS and Adios with different memory configurations. X% denotes the percentage of available local memory out of total memory use.

of throughputs. Thanks to the lightweight unikernel design, DiLOS outperforms a general-purpose design (Hermit), offering 42× better P99.9 latency given 0.7 MRPS throughput and 1.29× higher peak throughput. Adios has improved the P99.9 latency by 2.83× when DiLOS's latency starts to skyrocket as 1.3 MRPS. The peak throughput is 2.11×, 1.58×, and 1.59× better than Hermit, DiLOS, and DiLOS-P, respectively.

The source of Adios's improved latency and throughput over DiLOS is the dissolution of queuing delay induced by busy-polling. Figure 7(c) breaks down the latency of Adios when the throughput is 1.3 MRPS. Compared to DiLOS's latency breakdown in Figure 2(c), busy waiting has disappeared. The queuing delay has shrunk significantly: 16.3× (P99) and 36.8× (P99.9). This evaluation result is the first confirmation that Adios's yield-based page fault handling is effective.

In this evaluation, preemptive scheduling returns worse performance than busy-waiting page fault handling: DiLOS-P shows higher latency than DiLOS. With the local cache size to cover 20% of the working set, the request service time (without queueing delay) distribution is bimodal; 20% of local

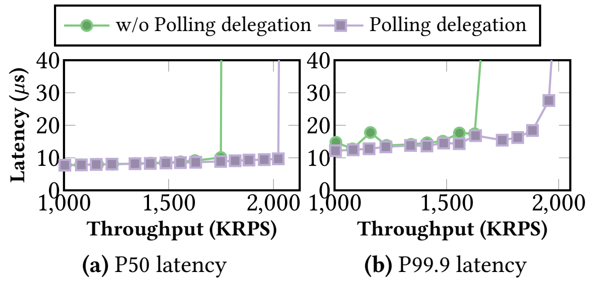


Figure 9. Latency comparison among TX mechanisms

cache access consumes 1.7 Kcycles or 0.85 μs and 80% of remote memory access consumes 10.6 Kcycles or 5.3 μs (Figure 2(c)). Thus a preemptive scheduler with an interval of 5 μs has very little room for enhancement and, even worse, carries penalty of preemption overheads.

Adios does not always have better latency than DiLOS because the yield-based page fault handler requires additional processing, such as context switches. When the throughput is low, Adios has slightly higher P50, P99, and P99.9 latencies than the busy-waiting page fault handler. However, their latency differences are only a few hundreds of *nanoseconds*. We conclude our yield-based PF handling brings in significant improvement in latency and its comparable overhead under low workload is justified.

Throughput. Figure 7(d) shows the throughput increase of DiLOS and Adios as the offered workload increases. While DiLOS's throughput stalls after 1.5 MRPS, Adios steadily increases until about 2.5 MRPS, providing 1.5× higher throughput. In Figure 7(e) we plot the RDMA utilization under the same setting as in Figure 7(d) and observe Adios' RDMA link utilization growing up to 82%.

These latency and throughput evaluation results in Figure 7 validate that *yield-based page fetching not only enhances latency but also throughput* by addressing HOL blocking issues and making full use of RDMA bandwidth.

Sensitivity to local DRAM size. In MD systems, local DRAM size is critical to performance. To examine how resilient Adios is to local DRAM size compared to DiLOS, we repeat the experiment with varying local DRAM sizes from 4GB (10% of total used memory) to unlimited (100%). Figure 8 shows its result. As local DRAM decreases from 100% to 10%, Adios has only a 25% throughput reduction, whereas DiLOS suffers from 60% degradation. Note that Adios with only 10% of local DRAM has a throughput similar to DiLOS with 80% of local DRAM. This result shows that when memory capacity is scarce, HOL blocking becomes a severe bottleneck, but yield-based page fault handling mitigates the impact. When available memory is unlimited, Adios offers slightly lower performance than DiLOS, from additional code path for yielding (*e.g.*, checking pages fetched before running new unithreads and maintaining a list for yielded unithreads). When no remote memory is used or no MD is configured, DiLOS whose code path is simpler achieves higher throughput than Adios.

Application	Type	Workload	Mem.	Modified
Memcached [14]	KVS	GET	40GB	71 LoC
RocksDB [24]	KVS	GET/SCAN	40GB	6 LoC
Silo [57]	OLTP	TPC-C [56]	20GB	24 LoC
Faiss [17]	VectorDB	BIGANN [27]	48GB	11 LoC

Table 2. Summary of real-world workloads

Effect of polling delegation. To evaluate the effect of polling delegation, we measure the median and P99.9 latency of Adios and Adios with polling delegation disabled. Without polling delegation, Adios transmits packets in a synchronous manner using busy-waiting. As shown in Figure 9, Adios has a 1.15× higher peak throughput compared to the one without polling delegation. For latency, at the peak throughput without polling delegation, 1,749 KRPS, polling delegation improves P99.9 latency 8.05×.

5.2 Real-World Application Study

To evaluate the performance of Adios in real-world applications, we compare Adios with baseline systems in Memcached [14], RocksDB [24], Silo [57], and Faiss [17] as summarized in Table 2. For all applications, we add remote memory flags in their mmap function calls. This modification requires only a few lines of code changes[6]. To link the applications and Adios' request handler, we add 100-300 LoC for *adapter* for each. The adapters parse requests and call related functions in their applications. For all experiments, the computing node has a local cache whose size is 20% of the total working set.

Memcached. We port Memcached v1.6.21 to Adios, replacing its original dispatcher and worker implementation [23] with those from Adios for compatibility. We conduct two experiments using value sizes of 128 and 1024 bytes. The key size was set to 50 bytes, with approximately 40GB of total memory usage and 8GB (20%) of local DRAM. Figures 10(a) to 10(d) present the tail and median latency for Memcached GET requests. For the 1024B workload, Adios achieves 1.60× better median latency and 5.18× better P99.9 latency at 730 KRPS compared to DiLOS. In the 128B workload, Adios outperforms DiLOS by 2.57× in median latency and 10.89× in P99.9 latency at 750 KRPS. In terms of throughput, Adios delivers 1.07× higher RPS for the 128B workload and 1.05× higher RPS for the 1024B workload than DiLOS. However, the improvements in throughput are modest in this experiment. The primary reason is that the NIC could not match the host's processing power, leading to longer RDMA queues that eventually reach full capacity. When the RDMA QPs are saturated, page fault handlers must pause, waiting for available slots in the QPs, which causes the dispatcher to pause and drop client requests. With the upcoming 200 and 400

[6]For Silo, we measure LoC from Caladan-variant.

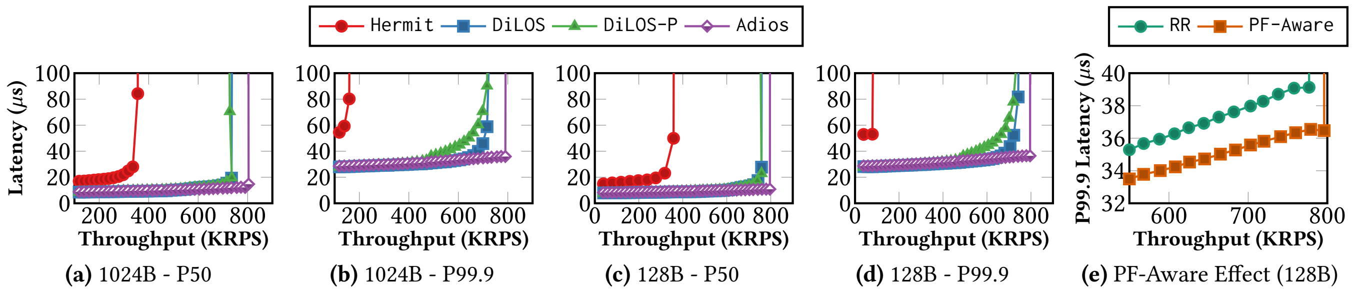


Figure 10. Memcached: P50 and P99.9 latency of 128B and 1024B GET workloads. For (e), 128B GET workload is used.

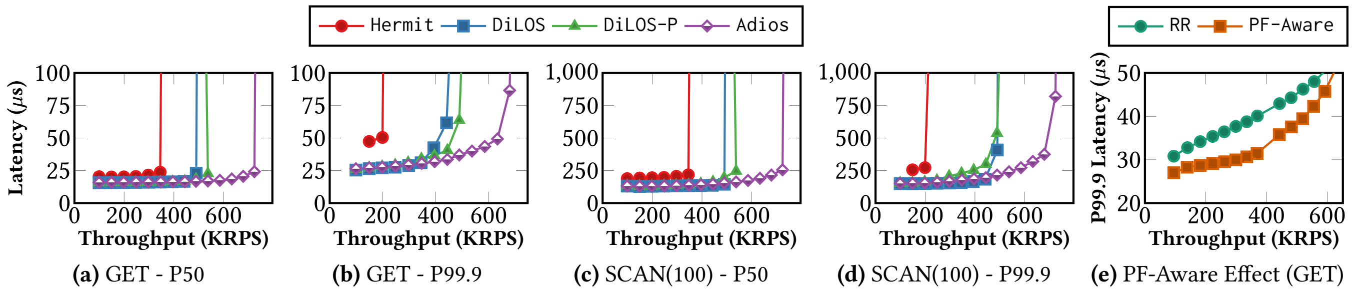


Figure 11. RocksDB: P50 and P99.9 latency of 99% GET and 1% SCAN(100) workload.

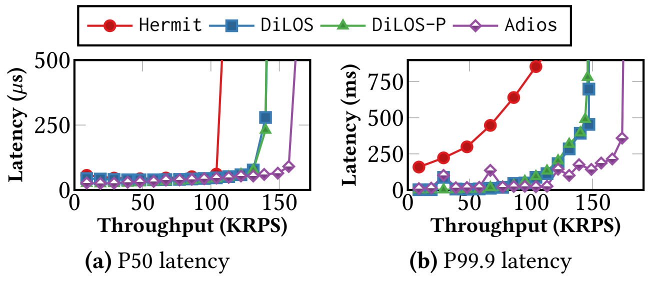


Figure 12. Silo: P50/P99.9 latency of TPC-C workload

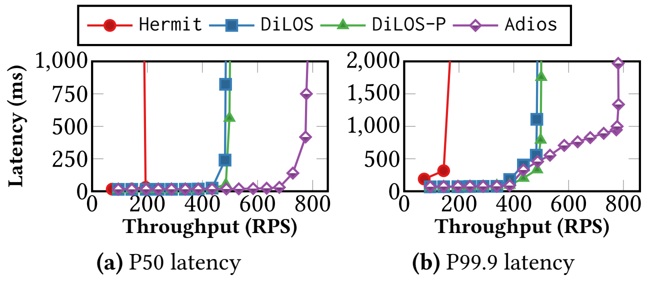


Figure 13. Faiss: P50/P99.9 latency of BIGANN workload

Gbps RNICs [13], Adios will be able to achieve significantly higher throughput without being constrained by the NIC.

RocksDB. We also conduct experiments with RocksDB, a popular in-memory key-value store with advanced features like scan operations. We select RocksDB because it has been included in evaluations in many previous studies [16, 28]. We use RocksDB v8.3.2 for the experiments. To optimize RocksDB for low latency, we use PlainTable [25] and mmap mode, which makes RocksDB read data from remote memory through load instructions and paging. We evaluate Adios's performance under the service time distribution with high dispersion, where the preemptive scheduler performs well. The load generator produces a bimodal workload, which consists of 99% of GET and 1% of SCAN(100) requests. The SCAN(100) request iterates over 100 keys starting from its argument and reads values referenced by the series of keys, thereby having a far longer service time. The exact ratio of SCAN(100)'s service time to GET's service time varies, for example, from 25× to 100× when the value size is 1024B, depending on the frequency of page faults triggered by the SCAN(100) request. Figures 11(a) to 11(d) show the median and P99.9 latency of GET and SCAN(100) requests. In DiLOS, a SCAN(100) request causes a long blocking of subsequent GET requests (HOL blocking), resulting in high P99.9 GET latency. For this workload, preemptive scheduling reduces the impact of HOL blocking. If processing a specific request takes a while (SCAN), DiLOS-P preempts its execution and handles other requests (mostly GET), improving the median and P99.9 latency of GET requests. Even though preemptive scheduling improves the latency of GET requests, Adios provides better performance in both latency and throughput. Overall, Adios offers 1.37×/7.61× better median/P99.9 GET latency when throughput is 490 KRPS and shows 1.47× better throughput than DiLOS. In comparison with DiLOS-P, Adios offers 1.33×, 2.71×, and 1.34× better median GET latency, P99.9 GET latency, and throughput, respectively.

Effect of PF-Aware Dispatching. To measure the effectiveness of our PF-aware dispatching algorithm, we compare our dispatcher (PF-Aware) with a round-robin dispatcher (RR) used in Shinjuku and Concord as a baseline. Figures 10(e) and 11(e) show the improvement introduced by PF-aware dispatching. Note that, for better readability, these two figures are magnified and their vertical axes do not start from zero. At all loads, PF-Aware has better P99.9 latency than RR. The maximum improvement is 7.5% for Memcached and 27% for RocksDB. These results confirm that PF-aware dispatching alleviates temporal page fault imbalance across workers and contributes to improved tail latency.

Silo. Silo is an in-memory transactional database (OLTP), and we use Caladan-variant Silo [20]. The Caladan-variant supports user-level threads, so we simply port it to Adios's unithreads. We extend Silo to support regular 4KB pages, as the original implementation only supports 2MB huge pages. Huge pages induces 512 times larger I/O amplifications than 4KB pages, seriously degrading page fetching latency. For the workload, we use the TPC-C benchmark with a scaling factor of 200 (about 20GB total working set in total) [56]. The benchmark consists of five request types in the following distribution: New-Order (44.5%), Payment (43.1%), Order-Status (4.1%), Delivery (4.2%), and Stock-Level (4.1%). Figure 12 shows the median and P99.9 latencies of Silo on each system. Adios has 4.66/3.85× and 2.24/2.26× better median and P99.9 latency than DiLOS/DiLOS-P at about 140 KRPS throughput. Adios also offers 1.67× higher throughput than Hermit and 1.18× higher throughput than DiLOS/DiLOS-P.

Faiss. Faiss is a library for similarity searching and clustering dense vectors. Vector similarity searching and clustering are widely used in datacenters to implement services such as recommendation, image retrieval, and text retrieval [17]. We use Faiss v1.8.0 to run an in-memory vector similarity search database. We utilize the IndexIVFFlat index in Faiss, which is the fastest indexing method but consumes a significant amount of memory [45]. To exhibit request-level parallelism in Adios, we opt-out of default OpenMP and use Adios's MD scheduler to manage concurrent tasks. For workload, we load and query on the vector DB using BIGANN dataset with 100M vectors (about 48GB memory consumption in total) [27]. Figure 13 shows the median and P99.9 latency of vector similarity search requests. Compared to other workloads, Faiss exhibits tens of milliseconds due to its complex search steps. When throughput is about 500 RPS, Adios shows 43.9/30.0× and 1.99/1.42× better median and P99.9 latency than DiLOS/DiLOS-P. Also, Adios offers 5.51×, 1.64×, and 1.58× more throughput than Hermit, DiLOS, and DiLOS-P. This result shows that Adios's design also improves systems whose request latency is tens or hundreds of milliseconds.

6 Discussion and Limitations

Assumptions and target use cases. The main source of performance improvement in Adios is from yield-based page fault handling: executing other tasks during remote page fetching. Therefore, Adios is beneficial if there are other tasks to run during page fetching. That is, the target uses cases of Adios are highly concurrent applications that use remote memory substantially. For example, in-memory databases, including key-value stores, OLTP systems, and vector DBs, are typical target applications for Adios. Moreover, RPC services that run memory-intensive business logic also can take advantage of Adios. On the other hand, compute-intensive or single or small threaded applications cannot gain performance from Adios. Compute-intensive workloads utilize a small amount of memory, and thus performance improvement in remote memory has a limited impact on overall performance. For single or small threaded applications, there are no or few unithreads to execute during page fetching, diminishing performance gains from yield-based page fault handing. Nevertheless, even with such poorly suited applications, Adios does not introduce a very significant performance degradation with respect to busy-waiting systems. For example, in the evaluations under low load (*e.g.*, throughput under 0.5 MRPS in Figure 7(a)), the number of concurrent threads to yield is zero or small, limiting the effect of yield-based page fault handling. Adios has only 6% of P99.9 latency increases compared to DiLOS in Figure 7(a) when throughput is 0.2 MRPS.

Networking protocol support. Adios' prototype currently supports UDP-based applications similar to Shinjuku and Perséphone. Networking protocol support is orthogonal to our design. However, we expect our design to be valid with TCP or other connection-oriented networking stacks if the networking stacks provide microsecond-scale latencies similar to IX [9], TAS [32], ZygOS [47], and Shenango [46]. We leave supporting more networking stacks as future work.

Limitations in scheduler. Adios is based on a cooperative scheduler and inherits the limitations from this approach. In particular, cooperative schedulers may not handle compute-intensive tasks fairly. Once a compute-intensive, long-running task is scheduled, it runs until completion, head-of-line blocking pending tasks. Also, the MD scheduler currently has limited scalability due to single queueing. As discussed in previous work [28], single queueing with a dedicated dispatcher thread can scale up to about ten worker cores. We leave improving Adios for better scalability as a future work.

Pinned threads. Adios employs two pinned threads (dispatcher and reclaimer) to guarantee low tail latency. However, thread pinning reduces the number of CPU cores for applications, reducing their achievable throughput. We leave designing a system without the pinned threads while offering tail latency increases as a future work.

7 Related Work

Memory disaggregation systems. There have been many MD systems based on a paging system [4, 21, 44, 48, 54, 61, 64], library or runtime [53, 59, 60, 67], or hardware features [11]. Infiniswap initially introduced paging for memory disaggregation and employed yield-based page fault handling. However, its kernel scheduler targets a few milliseconds scheduling, and its context-switching overhead is about 4 μs [40], incurring huge overheads. To hide the overheads, the paging-based systems have adopted busy-waiting for page fault handling. AIFM also points out the inefficiency of busy-waiting and implements MD as a library feature to avoid the inefficiency. Similar to ours, it also adopts lightweight threads and issues I/O asynchronously. However, since AIFM is implemented as a C++ library, it has limited portability, compatibility, and generality than Adios, a paging-based system. To the best of our knowledge, Adios is the first system that co-designs a scheduler and kernel to build yield-based page fault handling at a single-digit microsecond-scale.

Microsecond-scale schedulers. A series of studies have proposed microsecond-scale schedulers to address their challenges. ZygOS emphasizes the role of schedulers in handling microsecond-scale networked tasks and introduces a work-stealing scheduler, which achieves lower tail latency than parallel shared-nothing models [47]. Shinjuku demonstrates that single queue policy and preemptive scheduling further improve the tail latency when request service time distribution is highly dispersed [28]. Perséphone proposes a dynamic application-aware reserved cores (DARC) scheduling policy to guarantee low tail latency of short service time requests [16]. Shenango focuses on the CPU efficiency of a microsecond-scale scheduler and enhances the efficiency without sacrificing throughput or tail latency [46]. Caladan extends the Shenango scheduler to mitigate interference among applications through fast core allocation [20]. Instead of using preemption and work-stealing, Adios utilizes yield-based page fault handling to address HOL blocking problems in MD.

Unikernels and Library OSes. Adios is based on a unikernel, OSv [34]. However, any unikernel or library OS can be used to implement the design idea of Adios as long as it is based on a single address space and a single protection domain. There are many unikernels following the design [35–37, 43, 51]. In particular, stack designs of UKL (Unikernel Linux) and Adios share similar properties: kernel and user data are merged in a single stack [51]. However, since UKL utilizes the stacks of OS threads, each OS thread handles only one page fault concurrently and must be context switched to other OS threads to handle other requests. Adios, in contrast, stores the data into unithread's universal stack, and thus OS threads only need to switch unithreads to handle other requests, enabling multiple page fault handling within an OS thread. In addition to unikernels, several library OSes enable user-level applications to do paging for enhanced performance [8, 18, 22]. Exokernel allows applications to directly control hardware resources (*e.g.*, page tables), enabling application-specific customization of traditional operating systems [18]. Nemesis empowers the user-level scheduler to handle page faults for better Quality of Service (QoS) [22]. Dune leverages hardware virtualization features for performance and security improvements in user-level paging [8].

8 Conclusion

In this paper, we identify the performance pathologies of modern MD systems and design a novel MD system Adios. Adios implements a yield-based page fault handling with lightweight unithreads in unikernel and delivers microsecond-scale tail latency. Our evaluation demonstrates that Adios outperforms an existing state-of-the-art busy-waiting MD system (DiLOS) by up to 1.07×, 1.47×, 1.18×, and 1.64× in throughput; and 10.89×, 7.61×, 2.24×, and 1.99× in P99.9 latency for Memcached, RocksDB, Silo, and Faiss respectively.

Acknowledgement

We thank our shepherd Renaud Lachaize and anonymous reviewers for helpful comments. This work was partly supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No.RS-2022-NR070293 and No.RS-2024-00359979) and the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.RS-2024-00460762, Development of Core System Software for Fabric Memory-Based Computational Storage and No.RS-2024-00395134, DPU-Centric Datacenter Architecture for Next-Generation AI Devices).

References

[1] 2024. Postcopy. https://www.qemu.org/docs/master/devel/migration/postcopy.html.

[2] 2024. Userfaultfd. https://www.kernel.org/doc/html/next/admin-guide/mm/userfaultfd.html.

[3] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. 2014. CONGA: Distributed Congestion-Aware Load Balancing for Datacenters. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (Chicago, Illinois, USA) *(SIGCOMM '14)*. Association for Computing Machinery, New York, NY, USA, 503–514. doi:10.1145/2619239.2626316

[4] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can Far Memory Improve Job Throughput?. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 14, 16 pages. doi:10.1145/3342195.3387522

[5] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. 1991. Scheduler activations: effective kernel support for the user-level management of parallelism. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles* (Pacific Grove, California, USA) *(SOSP '91)*. Association for Computing Machinery, New York, NY, USA, 95–109. doi:10.1145/121132.121151

[6] Matan Barak. 2016. ibv_create_cq_ex - create a completion queue (CQ). https://man7.org/linux/man-pages/man3/ibv_create_cq_ex.3.html.

[7] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the Killer Microseconds. *Commun. ACM* 60, 4 (mar 2017), 48–54. doi:10.1145/3015146

[8] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe User-level Access to Privileged CPU Features. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 335–348. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/belay

[9] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 49–65. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay

[10] Sergey Blagodurov, Mike Ignatowski, and Valentina Salapura. 2021. The Time is Ripe for Disaggregated Systems. https://www.sigarch.org/the-time-is-ripe-for-disaggregated-systems/.

[11] CXL Consortium. [n. d.]. Compute Express Link. https://www.computeexpresslink.org.

[12] NVIDIA Corporation. 2023. Raw Ethernet Programming. https://docs.nvidia.com/networking/display/MLNXOFEDv531001/Programming.

[13] NVIDIA Corporation. 2024. ConnectX-7 400G Adapters. https://resources.nvidia.com/en-us-accelerated-networking-resource-library/connectx-7-datasheet.

[14] Inc. Danga Interactive. [n. d.]. Memcached. https://memcached.org.

[15] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (feb 2013), 74–80. doi:10.1145/2408776.2408794

[16] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. 2021. When Idling is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with PersépHone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 621–637. doi:10.1145/3477132.3483571

[17] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The Faiss library. (2024). arXiv:2401.08281 [cs.LG]

[18] D. R. Engler, M. F. Kaashoek, and J. O'Toole. 1995. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, Colorado, USA) *(SOSP '95)*. Association for Computing Machinery, New York, NY, USA, 251–266. doi:10.1145/224056.224076

[19] Linux Foundation. 2015. Data Plane Development Kit (DPDK). http://www.dpdk.org.

[20] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 281–297. https://www.usenix.org/conference/osdi20/presentation/fried

[21] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 649–667. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu

[22] Steven M. Hand. 1999. Self-Paging in the Nemesis Operating System. In *3rd Symposium on Operating Systems Design and Implementation (OSDI 99)*. USENIX Association, New Orleans, LA. https://www.usenix.org/conference/osdi-99/self-paging-nemesis-operating-system

[23] Holmes He. 2019. Understanding The Memcached Source Code - Event Driven I. https://holmeshe.me/understanding-memcached-source-code-VII/.

[24] Meta Platforms Inc. 2022. RocksDB. https://rocksdb.org.

[25] Meta Platforms Inc. 2024. PlainTable Format. https://github.com/facebook/rocksdb/wiki/PlainTable-Format.

[26] Rishabh Iyer, Musa Unal, Marios Kogias, and George Candea. 2023. Achieving Microsecond-Scale Tail Latency Efficiently with Approximate Optimal Scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 466–481. doi:10.1145/3600006.3613136

[27] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. 2011. Searching in one billion vectors: Re-rank with source coding. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 861–864. doi:10.1109/ICASSP.2011.5946540

[28] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 345–360. https://www.usenix.org/conference/nsdi19/presentation/kaffes

[29] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 437–450. https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia

[30] Naga Katta, Aditi Ghag, Mukesh Hira, Isaac Keslassy, Aran Bergman, Changhoon Kim, and Jennifer Rexford. 2017. Clove: Congestion-Aware Load Balancing at the Virtual Edge. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies* (Incheon, Republic of Korea) *(CoNEXT '17)*. Association for Computing Machinery, New York, NY, USA, 323–335. doi:10.1145/3143361.3143401

[31] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. 2016. HULA: Scalable Load Balancing Using Programmable Data Planes. In *Proceedings of the Symposium on SDN Research* (Santa Clara, CA, USA) *(SOSR '16)*. Association for Computing Machinery, New York, NY, USA, Article 10, 12 pages. doi:10.1145/2890955.2890968

[32] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. 2019. TAS: TCP Acceleration as an OS Service. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 24, 16 pages. doi:10.1145/3302424.3303985

[33] The kernel development community. [n. d.]. AF_XDP. https://www.kernel.org/doc/html/next/networking/af_xdp.html.

[34] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. 2014. OSv—Optimizing the Operating System for Virtual Machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 61–72. https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity

[35] 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. 2021. Unikraft: Fast, Specialized Unikernels the Easy Way. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 376–394. doi:10.1145/3447786.3456248

[36] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. 2020. A Linux in Unikernel Clothing. In *Proceedings of the Fifteenth European*

Conference on Computer Systems (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 11, 15 pages. doi:10.1145/3342195.3387526

[37] Stefan Lankes, Jens Breitbart, and Simon Pickartz. 2019. Exploring Rust for Unikernel Development. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems* (Huntsville, ON, Canada) *(PLOS'19)*. Association for Computing Machinery, New York, NY, USA, 8–15. doi:10.1145/3365137.3365395

[38] Jacob Leverich and Christos Kozyrakis. 2014. Reconciling High Server Utilization and Sub-Millisecond Quality-of-Service. In *Proceedings of the Ninth European Conference on Computer Systems* (Amsterdam, The Netherlands) *(EuroSys '14)*. Association for Computing Machinery, New York, NY, USA, Article 4, 14 pages. doi:10.1145/2592798.2592821

[39] Jialin Li, Naveen Kr Sharma, Dan RK Ports, and Steven D Gribble. 2014. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*. 1–14.

[40] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 49–64. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/litton

[41] Jiuxing Liu, Wei Huang, Bulent Abali, and Dhabaleswar K. Panda. 2006. High performance VMM-bypass I/O in virtual machines. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference* (Boston, MA) *(ATC '06)*. USENIX Association, USA, 3.

[42] H.J. Lu, Michael Matz, Milind Girkar, Jan Hubička, Andreas Jaeger, and Mark Mitchell. 2018. System V Application Binary Interface AMD64 Architecture Processor Supplement (With LP64 and ILP32 Programming Models) Version 1.0.

[43] Anil Madhavapeddy and David J. Scott. 2014. Unikernels: The Rise of the Virtual Library Operating System. *Commun. ACM* 57, 1 (jan 2014), 61–69. doi:10.1145/2541883.2541895

[44] Hasan Al Maruf and Mosharaf Chowdhury. 2020. Effectively Prefetching Remote Memory with Leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 843–857. https://www.usenix.org/conference/atc20/presentation/al-maruf

[45] Inc. Meta Platforms. 2020. Faster search. https://github.com/facebookresearch/faiss/wiki/Faster-search.

[46] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 361–378. https://www.usenix.org/conference/nsdi19/presentation/ousterhout

[47] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 325–341. doi:10.1145/3132747.3132780

[48] Yifan Qiao, Chenxi Wang, Zhenyuan Ruan, Adam Belay, Qingda Lu, Yiying Zhang, Miryung Kim, and Guoqing Harry Xu. 2023. Hermit: Low-Latency, High-Throughput, and Transparent Remote Memory via Feedback-Directed Asynchrony. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 181–198. https://www.usenix.org/conference/nsdi23/presentation/qiao

[49] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. 2018. Arachne: Core-Aware Thread Management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 145–160. https://www.usenix.org/conference/osdi18/presentation/qin

[50] Mubashir Adnan Qureshi, Yuchung Cheng, Qianwen Yin, Qiaobin Fu, Gautam Kumar, Masoud Moshref, Junhua Yan, Van Jacobson, David Wetherall, and Abdul Kabbani. 2022. PLB: Congestion Signals Are Simple and Effective for Network Load Balancing. In *Proceedings of the ACM SIGCOMM 2022 Conference* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 207–218. doi:10.1145/3544216.3544226

[51] Ali Raza, Thomas Unger, Matthew Boyd, Eric B Munson, Parul Sohal, Ulrich Drepper, Richard Jones, Daniel Bristot De Oliveira, Larry Woodman, Renato Mancuso, Jonathan Appavoo, and Orran Krieger. 2023. Unikernel Linux (UKL). In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 590–605. doi:10.1145/3552326.3587458

[52] Luigi Rizzo. 2012. netmap: A Novel Framework for Fast Packet I/O. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA, 101–112. https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo

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

[54] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 69–87. https://www.usenix.org/conference/osdi18/presentation/shan

[55] Timothy Stamler, Deukyeon Hwang, Amanda Raybuck, Wei Zhang, and Simon Peter. 2022. zIO: Accelerating IO-Intensive Applications with Transparent Zero-Copy IO. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 431–445. https://www.usenix.org/conference/osdi22/presentation/stamler

[56] TPC. [n. d.]. TPC-C. https://www.tpc.org/tpcc/.

[57] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 18–32. doi:10.1145/2517349.2522713

[58] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. 2003. Capriccio: scalable threads for internet services. *SIGOPS Oper. Syst. Rev.* 37, 5 (oct 2003), 268–281. doi:10.1145/1165389.945471

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

[60] Chenxi Wang, Haoran Ma, Shi Liu, Yifan Qiao, Jonathan Eyolfson, Christian Navasca, Shan Lu, and Guoqing Harry Xu. 2022. MemLiner: Lining up Tracing and Application for a Far-Memory-Friendly Runtime. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 35–53. https://www.usenix.org/conference/osdi22/presentation/wang

[61] Chenxi Wang, Yifan Qiao, Haoran Ma, Shi Liu, Wenguang Chen, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2023. Canvas: Isolated and Adaptive Swapping for Multi-Applications on Remote Memory. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 161–179. https://www.usenix.org/conference/nsdi23/presentation/wang-chenxi

[62] Peng Wang, Hong Xu, Zhixiong Niu, Dongsu Han, and Yongqiang Xiong. 2016. Expeditus: Congestion-Aware Load Balancing in Clos

Data Center Networks. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (Santa Clara, CA, USA) *(SoCC '16)*. Association for Computing Machinery, New York, NY, USA, 442–455. doi:10.1145/2987550.2987560

[63] Peter Xu. 2020. Userfaultfd-wp Latency Measurements. https://xzpeter.org/userfaultfd-wp-latency-measurements/.

[64] Wonsup Yoon, Jisu Ok, Jinyoung Oh, Sue Moon, and Youngjin Kwon. 2023. DiLOS: Do Not Trade Compatibility for Performance in Memory Disaggregation. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 266–282. doi:10.1145/3552326.3567488

[65] Hong Zhang, Junxue Zhang, Wei Bai, Kai Chen, and Mosharaf Chowdhury. 2017. Resilient Datacenter Load Balancing in the Wild. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Los Angeles, CA, USA) *(SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 253–266. doi:10.1145/3098822.3098841

[66] Yiwen Zhang, Yue Tan, Brent Stephens, and Mosharaf Chowdhury. 2022. Justitia: Software Multi-Tenancy in Hardware Kernel-Bypass Networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 1307–1326. https://www.usenix.org/conference/nsdi22/presentation/zhang-yiwen

[67] 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. 2022. Carbink: Fault-Tolerant Far Memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 55–71. https://www.usenix.org/conference/osdi22/presentation/zhou-yang