Revisiting Swapping in User-Space With Lightweight Threading

Kan Zhong, Wenlin Cui, Xin Chen, Qiao Li, Member, IEEE, Zhe Yang, Youyou Lu, Member, IEEE, Xiaodan Yan, Siwei Luo, Qizhao Yuan, and Keji Huang

Abstract—Memory-intensive applications, such as in-memory databases, caching systems, and key-value stores, are increasingly demanding larger main memory to fit their working sets. Conventional swapping can enlarge the memory capacity by paging out inactive pages to backend stores. However, existing swapping solutions suffer from several problems and compatibility issues, making them unsuitable for high-concurrency and memory-intensive applications. In this article, we redesign the swapping system and propose Lightswap, a high-performance user-space swapping solution that supports paging with both local SSDs and remote memories. First, to avoid kernel involvement, we propose to leverage the extended Berkeley packet filter (eBPF) for handling page faults (PFs) in user space and further eliminate the heavy I/O stack with the help of user-space I/O drivers. Then, we co-design the PF handling with lightweight thread (LWT) scheduling to improve system throughput and reduce the end-to-end PF latency. Finally, we propose a try-catch framework in Lightswap to deal with swap-in errors which have been exacerbated by the scaling in process technology. We implement Lightswap in our production-level system and evaluate it with various benchmarks. Results show that Lightswap achieves scalable PF notification latency (4 μs under 128 LWTs), reduces the PF handling latency by 3–5 times, and improves the throughput of memcached by more than 40% compared with the state-of-art swapping systems.

Index Terms—Extended Berkeley packet filter (eBPF), lightweight thread (LWT), memory disaggregation, user-space swapping.

I. INTRODUCTION

MEMORY-INTENSIVE applications [1], [2], such as in-memory databases, caching systems, and in-memory key-value stores, are increasingly demanding more and more memory to meet their low-latency and high-throughput requirements as these applications often experience significant performance loss once their working set cannot fit in memory. Therefore, extending the memory capacity becomes a mission-critical task for both researchers and system administrators.

Based on the virtual memory system, the existing OS provides swapping to enlarge the main memory by writing inactive pages to a backend store, which today is usually backed by SSDs. Compared to SSDs, DRAM still has orders of magnitude performance advantage, providing memory-like performance by paging with SSDs has been explored for decades [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13] and still remains great challenges. In particular, we have discovered that the heavy kernel I/O stack incurs significant performance overhead. When swapping in/out a single page, over 40% of the time is consumed by the I/O stack. Furthermore, this percentage will continue to rise if ultra-low latency storage media, such as Intel Optane and KIOXIA XL-Flash, are utilized as the backend store.

To avoid the high latency of paging with SSDs, memory disaggregation architecture [16], [17], [18], [19], [20] proposes to expose a global memory address space to all machines to improve memory utilization and avoid memory over-provisioning. However, the existing memory disaggregation proposals require new hardware supports [17], [18], [21], [22], [23], making them infeasible and expensive in real production environments. Fortunately, recent research, such as Infiniswap [24], Fluidmem [25], and AIFM [26], have shown that paging or swapping with remote memory is a promising solution to enable memory disaggregation. However, we still find the following critical issues of these existing approaches.

High Swap Latency: Kernel-based swapping, which relies on the heavy kernel I/O stack exhibits large software stack overheads, making it hard to fully exploit the high performance and low-latency characteristics of emerging storage media (e.g., Intel Optane) or networks (e.g., RDMA). We measured the remote memory access latency of Infiniswap, which is based on kernel swapping, as high as 40 μs even using one-side RDMA.

Low Scalability: Proposals like FluidMem [25] eliminate the slow kernel data path by handling page faults (PFs) in user space with the help of userfaultfd [27]. However, we observed that userfaultfd suffers from serious scalability issues under high concurrency. The measured PF notification latency of userfaultfd under 128 threads can be as high as 607 μs. Fastswap [28] improves kernel-based swapping by using a dedicated swap backend, but its poll waiting of critical page read lowers the CPU utilization and finally harms the scalability for high-concurrency applications.

2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.Authorized licensed use limited to the terms of the applicable license agreement with IEEE. Restrictions apply.
**Poor Compatibility:** New proposals like AIFM [26], which do not rely on OS’s virtual memory abstraction, provide a runtime-based swapping to applications and can largely reduce the I/O amplification. However, AIFM breaks the compatibility and require large efforts to rewrite existing applications.

Therefore, we argue that swapping needs to be redesigned to become high performance and remain good compatibility with existing applications. In this article, we propose Lightswap, a high-performance user-space swapping that supports paging with both SSDs and remote memories. First, Lightswap handles PFs in user space and utilizes the high-performance user I/O stack to eliminate the software stack overheads. To this end, we propose an ultralow latency PF handling mechanism to handle PFs in user space with the help of extended Berkeley packet filter (eBPF). When a PF happens, our dedicated eBPF program will switch the current context to the user-space PF handler to enable handling PFs in user space (Section IV-B). Second, after a PF happens, existing swapping solutions will block the faulting thread and hang its associated user requests, which lowers the system throughput and increases the response latency. To address this issue, Lightswap chooses to paging with lightweight thread (LWT) by leveraging its low scheduling cost. In Lightswap, LWTs are adopted to handle both user requests and PFs. We co-design the PF handling with LWT scheduling to make Lightswap achieve high performance and remain transparent to applications (Section IV-C). Note that Lightswap targets coroutine-based or multithreaded applications. These applications can be compatible with Lightswap by replacing their coroutines/threads with Lightswap’s LWTs.

Finally, storage/memory errors as well as network failures become commonplace in large-scale data centers, making both paging with local SSDs and remote memories become more prone to errors. As a result, pages in the backend stores may not be swapped back correctly; we refer this as swap-in errors. To tackle this issue, we propose a try-catch exception framework to enable Lightswap, the capability to tolerate swap-in errors without process termination (Section IV-D).

We evaluate Lightswap with memcached, MySQL, and index cache. Evaluation results show that Lightswap can reduce the PF handling latency by 3–5 times and improve the throughput by more than 40% when compared to Infiniswap. Besides, Lightswap can also outperforms AIFM with randomized access patterns. In summary, we make the following contributions.

1) We propose a novel user-space PF handling mechanism by utilizing eBPF. The proposed mechanism achieves ultralow PF notification latency (i.e., 4 µs under 128 threads), and outperforms both userfaultfd and signal approach by more than two orders of magnitude.

2) We propose to co-optimize user-space swapping with LWT, and demonstrate that the low switching overhead of LWT can contribute to achieving high performance in high-concurrency and memory-intensive systems.

3) We propose a try-catch-based exception framework to handle swap-in errors without process termination, thus improving the system’s availability.

4) We show the performance benefits of Lightswap with memcached, MySQL, and index cache, and compare it to other swapping systems.

The remainder of this article is organized as follows. Section II presents the background and motivation. Sections III and IV discuss our design considerations and the design details of Lightswap, respectively. Sections V and VI present the implementation and evaluation of Lightswap, respectively. We cover the related work in Section VI and conclude this article in Section VII.

II. BACKGROUND AND MOTIVATION

This section motivates Lightswap. We first introduce the traditional kernel-based swapping, followed by the reasons why we build the user-space swapping solution.

A. Kernel-Based Swapping

Swapping is an effective way to extend memory capacity by borrowing space from I/O devices (e.g., SSDs) in modern OSes [29]. In Linux kernel, a kernel thread named kswapd periodically monitors the free memory in system, once the number of free pages is below a preconfigured threshold, the kswapd thread tries to shrink in-use memory pages by writing (“swapping”) some inactive memory pages to some I/O devices (the “swap area”), making room for future memory allocations. In this process, page frame reclaim algorithm (PFRA) is employed to identify victim pages (i.e., inactive pages), and then these victim pages are written to I/O devices via a block interface. Since applications may get constantly stalled due to PFs if the request pages are swapped out, the latency of swapping in pages greatly determines the performance of applications. However, in conventional kernel-based swapping, pages that are swapped out/in must go through the whole storage stack to be written or read from the storage medium, making the I/O device driver and the underlying storage become an important factor on swap performance [30, 31].

B. Why User-Space Swapping

We build the user-space-based swapping solution for the following reasons:

**High-Performance User-Space I/O Drivers:** Existing swapping solutions [3, 4, 5, 6, 7, 8, 9, 10] depend on the existing kernel data path that is optimized for slow block devices, where both reading and writing pages from/to the backend store would introduce high software stack overheads. Recently, the performance of storage devices has been improved significantly due to the emerging technologies in both storage medium and interfaces, making the overhead of legacy kernel I/O stack becomes more and more noticeable. Fig. 1 compares the I/O latency breakdown of random read/write of Intel Optane SSD...
while using the kernel data path and the user-space SPDK [32] driver. The figure shows that nearly half (38.7%–48.3%) of the time is spent on the kernel I/O stack for both read and write while using the kernel data path. This overhead mostly comes from the generic block layer and device driver. In contrast, the I/O stack overhead is negligible while using the user-space SPDK driver. Therefore, user-space I/O drivers show high potential in improving swapping performance.

**High-Efficient Disaggregated Memory Support:** Disaggregated memory can be supported by paging with remote memories via the high bandwidth and low latency RDMA network. However, directly exposing remote memory as a local block device for paging in/out still requires going through the whole kernel I/O stack and introduces a high penalty in swapping latency. For example, we measured that the remote memory access latency in Infiniswap can be as high as 40 $\mu$s, which is about 10× higher than the latency of a 4-KB page RDMA read. User-space swapping not only has a much higher performance by eliminating the kernel stack, but also has more flexibility on remote memory management, caching eviction, and prefetching. For example, the swapping granularity (i.e., block size) can be customized on a per-process basis according to the application’s memory usage pattern. Therefore, we believe that user-space swapping has higher efficiency in enabling memory disaggregation.

### III. CHALLENGES AND DESIGN CONSIDERATION

To build a high-performance and practical user-space swapping, several challenges need to be resolved. This section discusses the design challenges of Lightswap.

#### A. Page Fault Notification

PFs need to be handled in user space to enable user-space swapping. The key challenge of handling PFs in user space is how to effectively notify the user-space PF handler when PFs happen. To achieve this, Linux provides userfaultfd [27] to notify and allow user-space applications to handle PFs of preregistered virtual address regions. Besides, one can also use the Linux signal to notify user-space applications that PFs happen. However, both userfaultfd and signal suffer from serious scalability issues as discussed below.

Userfaultfd requires a monitor thread that polls PF events from the userfaultfd file descriptor (i.e., uffd), and provides

**UFFD_COPY** and **UFFD_ZERO** ioctl operations to resolve the PFs. Fig. 2(a) shows how PFs are resolved with userfaultfd. When a PF happens, the OS PF handler puts the faulting thread into sleep and allocates a physical page for the faulted address, then an event is generated and sent to the monitor thread. The monitor thread reads the event from uffd and then wakes up a PF handling thread to resolve the PF. The PF handling thread will be responsible for reading the requested page and copying the page to the faulted address using **UFFD_COPY ioctl**. As shown in the figure, all the PF events are first added to the uffd and then polled by the monitor thread, making the contention on the uffd and monitor thread become the scalability bottleneck when multiple threads generate PFs simultaneously. Moreover, the context switching between the monitor thread and the PF handling thread also brings latency penalties in PF handling. Therefore, both the contention and context switching introduce serious scalability issues for userfaultfd in multithreading applications.

Fig. 2(b) shows the signal-based PF handling, when PFs happen, the OS PF handler notifies the user-space by sending a signal, which contains the faulted address and other related information. However, as the signal handler of a process is shared by all its threads, a lock is required to protect the signal handler data structure, which leads to serious lock contention under high concurrency.

To show the scalability issues of userfaultfd and signal, we record the PF notification latency (i.e., latency from the happening of PFs to the reception of such events by the user-space PF handler) of both userfaultfd and signal under different concurrent threads and plot their average latency in Fig. 3. As shown, with only one thread, both userfaultfd and signal achieve very low PF notification latency (i.e., 6 $\mu$s for userfaultfd and 1.7 $\mu$s for signal). However, with the increase of the number of concurrent threads, the PF latency of both userfaultfd and signal increases significantly. Therefore, neither userfaultfd nor signal is practical for handling PFs in the user space for high-concurrency applications, which motivates us to design a high-efficiency PF notification mechanism.

#### B. Swapping With Lightweight Thread

Traditionally, most high-concurrency applications use multithreading to handle user requests, and each thread is assigned multiple requests. However, with page swapping, PFs will block the thread that triggers the PF, leading to the user requests being hung up till the PF is resolved, which significantly reduces the system’s throughput and increases the...
response latency. Fig. 4 shows the impact of PFs on the request latency of a simple in-memory key-value system. In the tested KV system, we use two threads to handle user requests, and each thread has a work queue to buffer requests from concurrent clients. Fig. 4(a) gives the cumulative distribution function (CDF) of latency under different queue depths. As shown, PFs introduce large latency penalties in the overall latency of queued requests. Besides, we also observed that the tail latency is significantly increased due to PFs, as shown in Fig. 4(b). For example, the 90% latency increases by 5.6 times when the queue depth is 8.

To avoid blocking user requests due to PFs, one can launch one worker thread for each user request. However, each server may handle thousands of user requests concurrently, and launching one thread per request would lead to significant scheduling overhead, making most of the CPU time wasted on thread scheduling and context switching. Moreover, since PFs are handled in user space, it is difficult to reschedule the blocked thread to run in user space after the PF is resolved. To address these issues, LWT, also known as coroutine [33], [34], is introduced and widely adopted by memory-intensive applications to improve the throughput and reduce the response latency [35], [36]. Different from the OS-managed thread, such as pthread, LWT, is fully controlled and scheduled by a user-space program, instead of the OS. Each thread could contain lots of LWTs, and each LWT has an entry function that can suspend its execution and resume at a later point. Therefore, compared to OS-managed thread, LWT has a much lower scheduling overhead, which is a more flexible scheduling policy.

Lightswap mainly targets coroutine-based or multithreaded memory-intensive applications, which can be compatible with Lightswap with minor modifications. To effectively swap in pages in user space and make swapping be transparent to applications, we co-design PF handling with LWT scheduling (Section IV-C). First, when a PF is triggered by a normal application LWT, referred to as the faulting LWT, we will block the faulting LWT and launch a dedicated LWT, referred to as the swap-in LWT, to fetch pages from a backend store to the local memory. Second, the swap-in LWT will also be blocked and yields CPU for other normal application LWTs when waiting for data fetch. Finally, to reduce the overall PF latency, we adjust our LWT scheduler to prioritize swap-in LWTs, therefore the requested pages can be fetched as soon as possible.

C. Handling Swap-in Errors

Lightswap supports paging with both local SSDs and remote memories. Thus, swap-in errors happen when pages in the backend stores cannot be brought back to the local memory correctly due to storage, memory or network failures. To deal with this error, the common wisdom is to terminate the related process or even restart the whole system. Undoubtedly, this “brute force” method is simple and effective, but it also lowers the system’s availability and performance. For example, sudden failure of a memcached node can cause significant degradation of application performance because of cold cache. Previous research [37] has shown that the 95th response latency increases from 6 to 1600 ms and the performance loss takes more than 30 min to recover when memcached suffers from single node failure.

Fortunately, some applications, such as in-memory caching systems, can tolerate such data corruption/loss as they can recover data from disks or replicas. Therefore, in Lightswap, we propose an error handling framework, which provides an opportunity for applications to tolerate and correct swap-in errors using an application-specific error handling routine.

IV. LIGHTSWAP DESIGN

In this section, we first introduce the overview of Lightswap and its building blocks. Then we discuss how to effectively handle PFs in the user space and the co-design of swapping and LWT. Finally, we show how swap-in errors are handled in Lightswap.

A. Lightswap Overview

Fig. 5 illustrates the overall architecture of Lightswap. Lightswap handles PFs in user space and uses a key-value
interface for reading/writing pages from/to the backend store. The components of Lightswap are introduced as below.

**LWT Library:** An application can create multiple standard pthreads, which usually are bound to the given CPU cores. The number of pthreads is limited by the number of available CPU cores to minimize the scheduling overhead. Inside each pthread, LWTs are created to process user or client requests. LWT library is provided to user applications for creating and managing LWTs. In the LWT library, a scheduler is designed for scheduling LWTs based on priority. Different from the context switch of threads, which requires kernel involvement, the scheduling of LWTs is fully controlled by LWT library in user space without any kernel efforts. In our measurement, LWT switching latency is usually less than 1 µs, which is around 10 times faster than thread switching (several to tens of microseconds).

To handle the PFs triggered by LWTs, each pthread has a PF entry point (PF-entry) for the user-space PF handler. When a PF happens, this entry point will be reached and then the user-space PF handler will be involved (Section IV-B).

**Extension Memory Daemon (Etmemd):** etmemd is a user-space daemon for selecting victim pages of the given applications. To identify inactive pages accurately, etmemd utilizes a kernel module, namely, etmem to periodically clear and test the access bit in page table entries (PTEs). Before scanning PTEs, etmemd will first set the process ID and scanning interval for etmem through the `ioctl` interface. In our current implementation, we set the scanning interval to 20 s based on our experimental evaluation. Normally, the access bit in a PTE will be set to “1” by hardware once the page is touched. To count page hotness, etmem maintains an access counter for each page and increases the counter by one if the page’s PTE access bit is set in each scan. The scanning results (i.e., page access counts) are periodically reported to the user-space etmemd, which will sort pages by their access counts.

etmemd identifies victim pages based on both access count threshold and memory usage quota. If the physical memory usage quota of the given application is configured, etmemd selects inactive pages from the sorted page list for swapping out to ensure the memory usage quota is not exceeded. Otherwise, the pages whose access counts are below a predefined threshold are considered as inactive pages for swapping out.

**User-Space Swap Library (Uswaplib):** Uswaplib is the core component of Lightswap. It is a library that enables user-space swapping for applications. In the swap library, a dedicated swap-out thread is launched to receive victim pages from the etmemd and write them to the backend store. When swapping pages out, pages are first removed from the application’s page table using the `page_unmap()` system call and then added to the swap cache. Pages in the swap cache are asynchronously written back by a group of I/O LWTs, which periodically flush the swap cache to the backend store, thus decoupling page write from the critical path of the swap-out routine.

When a PF happens, the PF handler first searches the swap cache for the desired page. If the page is found in the swap cache, it is removed from the swap cache and added to the application’s page table at the faulted address. Otherwise, pages will be read from the backend store. In the PF handler, the prefetching is decoupled from the critical path of the swap-in routine. After bringing the desired page into memory, the PF handler appends a prefetch request into the prefetch queue. To improve the concurrency of page prefetching, a group of I/O LWTs, referred to as prefetchers constantly pull requests from the prefetch queue and bring pages into memory. Note that in our current implementation, we do not use any special or carefully designed prefetch algorithms as this work does not aim at prefetching. However, these algorithms can be easily added to Lightswap.

**eBPF prog:** The `eBPF prog` is the eBPF bytecode that is injected into the kernel using the `bpf()` system call. eBPF prog maintains two context maps: 1) default context map and 2) PF context map. Both maps contain multiple entries. Each entry of the default context map stores the default context, which is mainly comprised of CPU registers that are saved at `PF-entry`. Each entry of the PF context map stores the LWT context, which consists of CPU registers and faulted addresses that were captured at PF exceptions. When a PF happens, `eBPF prog` functions to 1) save the context of LWT into the PF context map and 2) switch the current context to the previously saved default context.

### B. Lightweight Page Fault Notification

The key challenge of handling PFs in user space is how to effectively notify the user-space PF handler when PFs happen. To achieve this, we propose the eBPF-based PF notification scheme. As shown in Fig. 6, our dedicated eBPF program hooks the kernel PF handling function (i.e., `do_page_fault()`), and it is used to bypass the kernel PF handler. In the application thread, before launching and scheduling LWTs, the thread enters the PF entry point (i.e., `PF-entry`) and saves the current thread context into the default context map (⃣). When one of its LWTs triggers a PF (⃣), the kernel PF handler will be invoked, and our eBPF program, which hooks `do_page_fault()` will also be executed. With the help of the eBPF program, a context switch

---

1 See our open source project: [https://gitee.com/openeuler/etmem](https://gitee.com/openeuler/etmem).
will be performed to make the thread jump to the user-space PF handler to resolve the LWT’s PF directly. In particular, in the eBPF program, we first save LWT’s context at the point that PF happens (❸). We refer this context as the PF context and store it in the PF context map, which uses the thread ID (i.e., tid) as the key and the LWT context as its value. The PF context, which contains the faulted address, will be used to restore the execution of faulting LWT after the PF is resolved.

Then, the thread’s default context (which is saved in step ❶) is retrieved from the default context map, and the current thread is switched to this context, making the current thread restored to PF-entry (❹). In PF-entry, the thread notices that the default context is already saved, which means this is not the first time of entering and thus a PF occurs. Therefore, the PF context is retrieved from the PF context map and saved in the faulting LWT’s stack. Then, the user-space PF handler is called to resolve the PF (❺).

In summary, as shown in Fig. 7, when an LWT triggers a PF, the eBPF program makes the thread switch to the PF handling LWT directly without interacting with other threads. Therefore, the resource contention in both userfaultfd and signal-based approaches are removed, making the proposed design achieve high scalability in PF handling.

C. Co-Design Paging Faulting Handling with LWT Scheduling

We co-design PF handling with LWT scheduling to improve the system throughput and reduce the end-to-end PF latency. Fig. 8 shows the pseudocode of how LWTs are scheduled and the user-space PF handler is called. In the application thread, after saving the default context, the LWT scheduler [i.e., LWT_sched()] continuously picks and executes LWTs from the front of the ready LWT queue. When a PF happens, the faulting LWT is blocked and the thread is restored to PF-entry(), in which the user-space PF handler is called. In order to make the application unaware of the PF, Lightswap resolves the PF transparently by creating a dedicated LWT (referred to as the swap-in LWT) to swap in the requested page in the user-space PF handler. To reduce the end-to-end PF latency, the LWT scheduler is designed to prefer swap-in LWTs and faulting LWTs. To achieve this, we classify the ready LWTs into three queues: 1) swap-in LWT queue, where swap-in LWTs reside after they are created; 2) faulting LWT queue, which contains ready LWTs that encounter PFs but their PFs have been resolved by swap-in LWTs; and 3) normal LWT queue, which contains other ready LWTs. The LWT scheduler assigns the highest priority to the swap-in LWT queue, the next is the faulting LWT queue, and the third is the normal LWT queue.

Authorized licensed use limited to the terms of the applicable license agreement with IEEE. Restrictions apply.
D. Try-Catch Exception Framework

Lightswap provides an exception framework for applications to handle swap-in errors in the user-space. As shown in Fig. 10, LIGHTSWAP_TRY and LIGHTSWAP_CATCH macros are embedded into application codes. Codes that are surrounded by the LIGHTSWAP_TRY macro are protected against swap-in errors. If the protected code triggers a PF but the requested page cannot be brought back to memory due to storage, memory, or network failures, the application LWT will jump to the LIGHTSWAP_CATCH immediately to handle the swap-in error using application customized routine, such as recover the data from the disk.

To achieve this, in the LIGHTSWAP_TRY macro, we save the context of the current LWT (❶), as shown in Fig. 10. When a swap-in error happens—the swap-in LWT cannot swap in requested pages correctly and terminates with an exception, the PF handler first save the faulted virtual address and then restores the LWT to LIGHTSWAP_TRY using the previously saved context (❷❸). In the LIGHTSWAP_TRY macro, the LWT checks if any swap-in error is encountered, if yes, the LWT will jump to the LIGHTSWAP_CATCH block to handle the swap-in error with application-specific error handling routine.

Note that the proposed try-catch framework gives the application an opportunity to address swap-in errors without application termination. For example, the in-memory caching system can return clients not found if the request data cannot be swapped back. However, not all the swap-in errors can be addressed properly. For example, if the corrupted/lost data belongs to the application’s metadata area, whose corruption has to restart the application.

V. IMPLEMENTATIONS AND COMPATIBILITY

A. Implementations

We implement and evaluate Lightswap in a real production system to show its effectiveness. We made some modifications to the Linux kernel to let it support user-space swapping. First, we add a hook point, more specifically, an empty function to the kernel. The kernel PF handler (i.e., do_page_fault()) will call this empty function if the faulted address belongs to an application that is supported by Lightswap. We hook the eBPF program to this empty function so that normal PFs still use the kernel PF handler, only applications that are supported by Lightswap use our user-space PF handler. To reduce the number of bpf() system calls, we use shared memory between kernel and user-space to share the PF context map.

Second, to support swapping in/out pages in the user space, we added a pair of system calls (i.e., page_map() and page_unmap()) to, respectively, map or unmap a page to or from a given virtual address. Totally, all these kernel modification efforts are no more than 500 lines of code. To avoid adding new system calls to the kernel, we will study how to enable swapping in the user space using the standard system calls, such as mmap(), unmap(), and madvice() in our future work.

We implement a key-value store as the backend store for swapping. Fig. 11 shows the I/O models used in the backend store. For NVMe SSDs, the backend store employs the SPDK-based user-space driver to achieve the best performance. To keep good compatibility with existing systems, the backend store also supports interacting with SSDs via the kernel-based io_uring interface. For remote memories, the backend store communicates with remote servers with our customized user-space RDMA driver. Since the swap-in operations are in the critical path of PFs, the backend store checks for completed I/O by polling. To accelerate paging out, we implement a batch put operation that can write pages in batches to the key-value store. Therefore, we organize the SSD space in a log-structured way and thus pages in the swap cache can be flushed in batches to maximize the throughput. For remote memory, we deploy a daemon in remote memory servers to reserve and allocate memory space. To reduce the number of allocation requests, memory servers only allocate 1-GB memory blocks and return to clients the registered memory region IDs and offsets of memory blocks for RDMA. The backend store in the client manages memory blocks and splits them into pages.

B. Compatibility Discussions

Lightswap mainly targets at high-concurrency and low-latency memory-intensive applications, where coroutines are widely adopted to improve the throughput and reduce the response latency. To reduce the application rewritten overhead, we provide applications an LWT library and a swap library that, respectively, encapsulates the LWT scheduler and user-space PF handling routine. To be compatible with Lightswap, applications only need to replace their coroutines

Authorized licensed use limited to: BOSTON UNIVERSITY. Downloaded on July 23,2023 at 04:19:25 UTC from IEEE Xplore. Restrictions apply.
with LWTs by using the APIs provided in our LWT library. The application’s logic and algorithm remain unchanged, and the developers also do not need to care how the PFs are handled and LWTs are scheduled.

Lightswap also supports conventional multithreaded applications. Thread-based applications need to launch LWTs inside each thread to handle user requests instead of the thread itself. In the evaluation, we have rewritten memcached, which is a multithreaded in-memory caching system, to use our LWT library and enable user-space swapping. To achieve this, for each worker thread of memcached, we create LWTs inside each thread to handle client requests and keep its caching logic unchanged.

To summarize, although Lightswap requires modifications to both kernel and applications, the modification efforts are acceptable and can bring significant performance improvement (e.g., 60% higher throughput for memcached) compared to the default Linux swap, which does not need any modification.

VI. Evaluation

This section introduces the evaluation of Lightswap. It starts with a brief introduction of our system implementation, then gives the evaluation setups, and finally discusses the results.

A. Evaluation Setups

We employ two X86 servers in the evaluation, one as the client for running applications, and the other as the memory server to allocate memory blocks. Each server equips with two Intel Xeon CPUs, and each CPU contains 40 cores with hyper-threading disabled. The memory capacity of both servers is 256 GB. We will limit the memory usage of the client server in order to trigger swapping in/out. The connection between the two servers is 100G RoCE with our customized user-space RDMA driver. We use KIOXIA XL-Flash as the locally PCI-attached NVMe SSD. In our measurements, the raw latency of XL-Flash is around 25 μs and the latency of one-side RDMA read for 4 KB is around 5 μs. We compare Lightswap to both the Linux kernel swapping (based on kernel version 5.17) and other disaggregated memory systems. For fair comparison and to minimize the impacts of the swap cache on performance, we limit the size of the swap cache to 1 GB for all swapping schemes.

B. Microbenchmarks

1) Page Fault Notification Latency: To show the effectiveness of eBPF-based user-space PF handling, we first evaluate the average PF notification latency under different degrees of concurrency, where the PF notification latency denotes the latency from PF happening to the user-space PF handler receiving the PF event. Since it is hard to accurately measure the frequency of resource contention (i.e., the number of simultaneous PFs) in PF handling, we use this metric to show how the contention contributes to the PF handling latency. We set the physical memory ratio to 50% and use a randomized memory access pattern. Therefore, the probability of PF is about 50%, and about 50% concurrent threads would suffer from contention.

2) Page Fault Handling Latency: To show the end-to-end performance of Lightswap, we compare the PF handling
latency of Lightswap to other swapping schemes. The page
handling latency is denoted as the time duration from the
PF happening to the PF handler finishing resolving the PF.
We compare the results between Lightswap, Infiniswap,
and the Linux default kernel swap. Fig. 12 illustrates the
evaluation results. Note that PF handling latency does not include
the time duration from the PF being resolved to the point
when the faulting thread/LWT is restored to run. As shown,
when paging with remote memory through one-side RDMA,
Lightswap achieves the lowest PF handling latency, ranging
from 10 to 13.5 µs under different degrees of concurrency.
The conventional Linux kernel swap has the highest PF han-
dling latency, ranging from 43.2 to 63.2 µs. When paging
with remote memory via one-side RDMA, Lightswap, respectively,
outperforms Infiniswap and Linux kernel swap by around 2.5–
3.0 times and 4.3–5.0 times in terms of PF handling latency.
Since io_uring reduces the number of syscalls and uses polling
for I/O completion, for paging with local SSD, io_uring almost
achieves the same performance compared to the SPDK. Both
of io_uring and SPDK version Lightswap have comparable
performance with Infiniswap, and have around 30% lower
latency than the Linux kernel swap.

To demystify the reason behind this improvement, we break
the PF handling process and plot its time consumption in
Fig. 13. In the figure, the kernel PF handler has already
included the time spent on trapping into the kernel. Lightswap
handles PFs in user space and avoids the slow kernel data
path by leveraging high-performance I/O libraries. For both
paging with remote memory and local SSD, the PF handling
latency of Lightswap is dominated by the page fetch latency. In
particular, the software stack of Lightswap-SSD only counts
for 13.09%. Although io_uring still needs to go through all
the kernel layers, it avoids expensive I/O scheduling, reduces
the number of syscalls, and uses polling for I/O completion.
Thus, Lightswap-SSD-iou only increases the latency percent
of the software stack lightly. We believe that Lightswap-
SSD-iou can have lower PF handling latency when using the
CPU-consuming kernel thread polling mode. In the follow-
ing evaluation, we only use Lightswap-SSD for performance
comparison since the SPDK-based I/O model exhibits better
performance compared to io_uring.

For Infiniswap and Linux kernel swap, both of which need
to go through the entire heavy kernel I/O stack when fetching

Fig. 12. Average PF handling latency. Lightswap-RDMA denotes paging with
remote memory via RDMA, Lightswap-SSD represents paging with local SSD
via SPDK-based user-space driver, and Lightswap-SSD-iou represents paging
with local SSD via kernel io_uring interface.
Fig. 13. PF handling latency breakdown under no concurrency. The numbers beside each bar denote the fraction of time cost by the software stack during handling PFs.

Fig. 14. Memory footprint control result.

(i.e., 80 in total), all the worker threads/LWTs can occupy the CPU continuously. Therefore, we observed that there is no performance difference between LWT-version and thread-version memcached in the baseline. The following three key results are drawn from these figures: 1) Lightswap-RMDA has the least performance degradation, and its throughput outperforms Infiniswap and Linux kernel swap by 40% and 60% on average, respectively; 2) due to the outstanding PF handling latency, Lightswap-RDMA achieves the lowest latency among all swapping schemes, and it outperforms Infiniswap and Linux kernel swap by 18% and 30% on average, respectively; and 3) despite the higher operation latency, Lightswap-SSD still achieves 10%–20% higher throughput than Infiniswap. Thanks to the LWT, in LWT-version memcached, instead of blocking the current worker thread, PFs only block the faulting LWTs. Thus, other worker LWTs can still get scheduled to process clients’ requests, resulting in achieving higher throughput. In contrast, in the thread-version memcached, the current worker thread will be blocked once a PF happens, as well as the user requests assigned to that worker thread. Therefore, as shown in Fig. 17, both Lightswap-SSD and Lightswap-RDMA have much lower medium and 99th percentile latency than Infiniswap.

Swap-In Error Handling: In order to show the effectiveness of the proposed swap-in error handling framework, we generate random swap-in errors for the address space used by memcached. Currently, we add a simple swap-in error handling routine in do_item_get() of memcached. In memcached, items belonging to the same slab class are linked in the same LRU list. If the error handling routine finds that any item cannot be swapped back due to swap-in error, it has to reset the whole slab class and returns “not found” as the response to the GET requests. However, if a swap-in error causes any corruption in the memcached metadata, such as the hash table and slab class array, the error handling routine has to terminate memcached.

Table II shows the results of simulated swap-in error handling results for memcached workload. We generate 10 thousand swap-in errors to memcached. As we only protect the GET operation, the readmost workload exhibits the highest survival rate, and more than half of the swap-in errors are handled properly without process termination. On the contrary, the writemost workload, which is dominated by PUT operations makes memcached get killed in 97% of the swap-in errors. We believe that with more try-catch protections, memcached has more chance to survive when suffering swap-in errors.

2) MySQL: We measure the performance of MySQL with benchmarkSQL under different numbers of warehouses. BenchmarkSQL tests MySQL with the TPC-C standard, which queries orders, users, and products randomly. Since MySQL handles clients’ requests with a thread pool and new threads will be created when it finds worker threads are blocked while the request queue is not empty, we keep the thread mode unchanged and only add Lightswap’s eBPF-based PF notification mechanism and user-space PF handling routine to MySQL to simplify the evaluation.

In the baseline, we configure the maximum innoDB buffer pool size to 35 GB and find that the physical memory usages (vm_RSS) of MySQL are 18.3 and 44.3 GB when the warehouses are 100 and 1000, respectively. In the evaluation, we therefore, respectively, limit the physical memory usages of MySQL to 9 and 22 GB under warehouse 100 and 1000, making around 50% working set reside in memory. Fig. 18 compares the transactions per minute (tpmC) of MySQL under different swapping solutions. As shown, Lightswap-RDMA achieves the lowest performance loss, which is around 2%–3% compared to the baseline. Infiniswap has a 10%–22% performance loss compared to the baseline. During the evaluation of Lightswap, we find that the physical memory sizes used by MySQL are, respectively, 8.7 and 21.3 GB under
warehouses 100 and 1000. When the warehouse is 1000, Lightswap-RDMA outperforms Infiniswap by around 25%, and Lightswap-SSD achieves nearly the same performance as Infiniswap. In this test, since the TPC-C performance of the database is not strongly sensitive to the buffer pool size when the ratio of buffer pool size to data size exceeds 10% [39], we observe that the Linux kernel swap also achieves acceptable throughput, which only has 18%–32% performance degradation compared to the baseline.

3) Index Cache: In storage systems, such as network-attached storage (NAS) and storage-attached network (SAN), index cache performs a key role in reducing the read/write latency. In our SAN system, we adopt the LSM tree as the persistent index structure, which is stored as index blocks in disks. The index cache is organized as an in-memory hash table that maintains an LSM-tree index block to accelerate index lookup. In the evaluation, we configure the index block size to 512 bytes. The total size of our LSM-tree index structure is 42 GB, and we use the fully cached index as the baseline. We compare the index lookup performance under different configurations. For AIFM [26], we use a block-granular LRU list to guide the memory evacuator to reclaim memory space. For Lightswap, we swap pages to remote memory via RDMA.

Figs. 19 and 20 show our evaluation results for both skewed and randomized access distribution. We observe the following findings. First, memory miss introduces a large penalty for

---

**Fig. 15.** Average throughput per-second (TPS) of memcached with different swapping solutions. We use 32 worker threads to process requests for all these configurations. "Baseline" indicates all datasets reside in memory; "75%," and "50%," respectively, indicate 75% and 50% datasets reside in memory. (a) Readmost (Read: 90%, Insert/Update: 10%). (b) Readwrite (Read: 50%, Insert/Update: 50%). (c) Writemost (Read: 10%, Insert/Update: 90%).

**Fig. 16.** Comparison of average memcached operation latency under different swapping solutions. (a) Readmost. (b) Readwrite. (c) Writemost.

**Fig. 17.** Comparison of medium and 99th percentile operation latency for readwrite workload. (a) Medium. (b) 99th percentile.
randomized index lookup, making all the swapping schemes exhibit high latency in index lookup. Lightswap has the lowest randomized index lookup latency but the latency still increased from 2 to 28 µs and 93 µs when the physical memory watermarks are, respectively, 75% and 50% under 16 concurrent LWTs. Second, all swapping solutions can achieve acceptable performance for skewed index lookup compared to the baseline due to higher cache efficiency. The skewed index lookup has up to 32× lower latency than the randomized index lookup. For example, in Fig. 20, the index lookup latency of AIFM reduced from 163 to 5 µs when the access distribution changed from random to skew under 75% physical memory. Third, due to the memory resource contention, higher concurrency will lead to much longer lookup latency, especially for randomized access distribution. Finally, AIFM exhibits the best performance in skewed index lookup under a high (75%) physical memory watermark, but it has the highest latency under randomized access or a low (50%) physical memory watermark. This phenomenon mainly comes from two factors: 1) high lock contention between worker threads and evacuator threads due to high memory miss rate in randomized access or low physical memory watermark and 2) embedded removable pointer invalidation needs additional CPU cycles when moving objects to remote memory. Because of these factors, AIFM does not perform well in high-concurrency and high-memory contention scenarios. Differently, Lightswap utilizes PFs to achieve transparently paging with remote memory and is optimized for high concurrency with the help of LWT.

VII. RELATED WORK

SSD-Based Swapping: Swapping has been studied for years, with performances of magnitude improvements compared to hard disks. SSDs-based swapping becomes an attractive solution to extend the effective memory capacity. To this end, kernel-based swapping has been revisited and optimized for SSDs [3, 4, 5, 6, 7, 8, 9, 10] to enlarge the main memory. They are integrated with Linux virtual memory and rely on the paging mechanism to manage the page movement between host DRAM and SSDs. Different from these application-transparent approaches, runtime-managed and application-aware swapping schemes [11, 12, 13] are proposed to fully exploit the performance of flash memory and alleviate the I/O amplification. However, all these swapping schemes, including both OS-managed and runtime-managed approaches, employ kernel-level SSD drivers, and thus I/O traffic needs to go through the whole storage stack, which may introduce notable software overheads as the next-generation storage technology like Intel Optane [14] and KIOXIA XL-Flash [15] are much faster than the past ones.

Disaggregated and Remote Memory: Several works [40, 41, 42, 43, 44, 45, 46] have already explored paging with remote memory instead of local SSDs, but their performance is often restricted by the slow networks and high CPU overheads. With the support of RDMA networks and emerging hardware, it is possible to reorganize resources into disaggregated clusters [16, 17, 18, 19, 20, 47, 48, 49] to improve and balance the resource utilization. To achieve memory disaggregation, Fastswap [28] and INFINISWAP [24] explore paging with remote memory with kernel-based swapping. FluidMem [25] supports full memory disaggregation for virtual machines through hotplug memory regions and relies on userfaultfd to achieve transparent PF handling. AIFM [26] integrates swapping with application and operates at object granularity instead of page granularity to reduce network amplification. Semeru [50] provides a JVM-based runtime to managed applications with disaggregated memory and offloads garbage collection to remote memory servers. Remote regions [51] applies file abstraction to remote memory and provide both block and byte access interfaces. In this article, we implement a user-space swapping solution and co-design it with LWT for data-intensive and high-concurrency applications.

Distributed Shared Memory (DSM): DSM systems [52, 53, 54, 55, 56, 57] provide a unified abstraction by exposing a shared global address space to applications. Different from remote memory, DSM provides a memory abstraction in that data is shared across different hosts, therefore bringing significant cache coherence costs and making DSM inefficient. To avoid the coherence costs, partitioned global address space (PGAS) [58, 59, 60, 61] is proposed but it requires application modification. Lightswap is more efficient as it allows applications to transparently paging with remote memory.

VIII. CONCLUSION

This article proposes a user-space swapping mechanism that can fully exploit the high performance and low latency of emerging storage devices, as well as the RDMA-enabled remote memory. We focus on three main aspects: 1) how to handle PFs in the user space effectively; 2) how to reduce the end-to-end PF handling latency and improve the throughput of user-space swapping; and 3) how to make applications tolerate swap-in errors with customized error handling routine.

REFERENCES

[1] “Memcached.” 2021. [Online]. Available: https://memcached.org/
[2] “VoltDB.” 2021. [Online]. Available: https://www.voltdb.com/
[3] M. Saxena and M. M. Swift, “FlashVM: Virtual memory management on flash,” in Proc. USENIX Annu. Tech. Conf. (ATC), 2010, p. 14.
[4] S. Ko, S. Jun, Y. Ryu, O. Kwon, and K. Koh, “A new Linux swap system for flash memory storage devices,” in Proc. Int. Conf. Comput. Sci. Appl., 2008, pp. 151–156.
[5] S.-Y. Park, D. Jung, J.-U. Kang, J.-S. Kim, and J. Lee, “CFLRU: A replacement algorithm for flash memory,” in Proc. Int. Conf. Compilers, Archit. Synth. Embedded Syst. (CASES), 2006, pp. 234–241.
[6] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi, “Operating system support for NVM+DRAM hybrid main memory,” in Proc. 12th Conf. Hot Topics Oper. Syst. (HotOS), 2009, pp. 1–5.
[7] A. Abulilia et al., “FlatFlash: Exploiting the byte-accessibility of SSDs within a unified memory-storage hierarchy,” in Proc. 24th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS), 2019.
[8] V. Fedorov, J. Kim, M. Qin, P. V. Gratz, and A. L. N. Reddy, “Speculative paging for future NVM storage,” in Proc. Int. Symp. Memory Syst. (MEMSYS), 2017, pp. 399–410.
[9] J. Huang, A. Badam, M. K. Qureshi, and K. Schwanz, “Unified address translation for memory-mapped SSDS with flashmap,” in Proc. 42nd Annu. Int. Symp. Comput. Archit. (ISCA), 2015, pp. 580–591.
[10] N. Y. Song, Y. Son, H. Han, and H. Y. Yeom, “Efficient memory-mapped I/O on fast storage device,” ACM Trans. Storage, vol. 12, no. 4, p. 19, 2016.
D. Liu, K. Zhong, X. Zhu, Y. Li, L. Long, and Z. Shao, “Non-

Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay, “

J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, “Efficient 

I. Calciu et al., “Rethinking software runtimes for disaggregated 

K. Lim et al., “System-level implications of disaggregated memory,” in 

F. Li, S. Das, M. Syamala, and V. R. Narasayya, “ Accelerating relational databases by leveraging remote memory and RDMA,” in Proc. Int. Conf. Manage. Data (SIGMOD), 2016, pp. 355–370.

P. S. Rao and G. Porter, “Is memory disaggregation feasible? A case study with spark SQL,” in Proc. ACM/IEEE Symp. Architect. News. Commun. Syst. (ANC5), 2016, pp. 75–80.

“ The machine: A new kind of computer.” HP, 2021, [Online]. Available: https://www.hpl.hp.com/research/systems-research/themachine/

K. Lim et al., “System-level implications of disaggregated memory,” in Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA), 2012, pp. 1–12.

I. Calciu et al., “Rethinking software runtimes for disaggregated memory,” in Proc. ASPLOS, 2021, pp. 79–92.

J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, “Efficient memory disaggregation with inifiniswap,” in Proc. 14th USENIX Symp. New. Sytem. Design. Implement. (NSDI), 2017, pp. 649–667.

B. Caldwell, “FluidMem: Open source full memory disaggregation,” Ph.D. dissertation, Dept. Comput. Sci., Univ. Colorado Boulder, Boulder, CO, USA, 2019.

Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay, “[AIFM]: High-performance, application-integrated far memory,” in Proc. 14th (USENIX) Symp. Oper. Syst. Design. Implement. (OSDI), 2020, pp. 315–332.

“Userfaultd,” The Kernel Development Community, 2021. [Online]. Available: https://www.kernel.org/doc/html/latest/admin-guide/mm/userfaulthud/index.html

E. Amaro et al., “Can far memory improve job throughput?” in Proc. 15th Eur. Conf. Comput. Syst. (EuroSys), 2020, p. 14.

W. Maurer, Professional Linux Kernel Architecture. Birmingham, U.K.: Wrox Press, 2008.

D. Liu, K. Zhong, X. Zhu, Y. Li, L. Long, and Z. Shao, “Non-volatile memory based page swapping for building high-performance mobile devices,” IEEE Trans. Comput., vol. 66, no. 11, pp. 1918–1931, Nov. 2017.

K. Zhong et al., “Building high-performance smartphones via non-volatile memory: The swap approach,” in Proc. Int. Conf. Embedded Softw. (EMSOFT), 2014, pp. 1–10.

[Intel Corp., Santa Clara, CA, USA]. Storage Performance Development Kit, 2021. [Online]. Available: https://spdk.io

M. E. Cooigue, “Design of a separable transition-diagram compiler,” Commun. ACM, vol. 6, no. 7, pp. 396–408, 1963.

A. L. D. Moura and R. Jersjalsmschy, “Revisiting coroutines,” ACM Trans. Program. Lang. Syst., vol. 31, no. 2, pp. 1–31, 2009.

C. Jonathan, U. F. Minhas, J. Hunter, J. Levandoski, and G. Nishanov, “Exploiting coroutines to attack the killer nanoseconds,” in Proc. VLDB Endow., vol. 11, no. 11, pp. 714–721, 2018.

G. Papapoulos, T. Lefler, N. May, and A. Alilamik, “Interleaving with coroutines: A practical approach for robust index joins,” Proc. VLDB Endow., vol. 11, no. 2, pp. 230–242, 2017.
Kan Zhong received the B.S. and Ph.D. degrees in computer science from the College of Computer Science, Chongqing University, Shanghai, China, in 2013 and 2018, respectively.
He was a Visiting Scholar with the Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL, USA, from 2016 to 2017. He is currently a Research Engineer with Huawei Inc., Shenzhen, China. His research interests include storage systems, emerging nonvolatile memory, and computer architecture.

Youyou Lu (Member, IEEE) received the B.S. degree in computer science from Nanjing University, Nanjing, China, in 2009, and the Ph.D. degree in computer science from Tsinghua University, Beijing, China, in 2015.
He is an Associate Professor with the Department of Computer Science and Technology, Tsinghua University. His research works have been published at a number of top-tier conferences, including FAST, OSDI, SOSP USENIX ATC, EuroSys, and SC. His research interests include file and storage systems spanning from architectural to system levels.
Dr. Lu’s research won the Best Paper Award at NVMSA 2014 and was selected into the Best Papers at MSST 2015.

Wenlin Cui received the B.S. degree from the Chongqing University of Posts and Telecommunications, Chongqing, China, in 2005.
He has more than 17 years of Research and Development experience. He is currently engaged in storage system architecture research and development with Huawei Inc., Shenzhen, China. His research interests include memory-based storage systems, hardware/software co-design, and large-scale distributed systems.

Xiaodan Yan received the M.S. degree from Xidian University, Xi’an, China, in 2009.
She is currently an Operating System Engineer with Huawei Inc., Shenzhen, China. She has been engaged in the Research and Development of Operating System in the server field for more than ten years. Her research interests include storage systems and emerging nonvolatile memory technologies.

Xin Chen received the B.S. degree in physics from the University of Electronic Science and Technology of China, Chengdu, China, in 2019.
He is currently a Senior Engineer with Huawei Inc., Shenzhen, China. His research interests include storage systems and emerging in-memory key-value store.

Siwei Luo received the B.S. and Ph.D. degrees from the School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu, China, in 2012 and 2017, respectively.
He is currently a Research Engineer with Huawei Inc., Shenzhen, China. His research interests include distributed storage, nonvolatile memory, memory fabric, and computer architecture.

Qiao Li (Member, IEEE) received the Ph.D. degree from the Department of Computer Science, City University of Hong Kong, Hong Kong, in 2021.
She is currently an Associate Professor with the School of Informatics, Xiamen University, Xiamen, China. Her research interests include NAND flash memory, storage systems, and computer architecture.

Qizhao Yuan received the Ph.D. degree in computer science from the Chinese Academy of Sciences, Beijing, China, in 2015.
He is currently a System Engineer with Huawei Inc., Shenzhen, China. His research interests include storage systems and computer architecture.

Zhe Yang was born in 1996. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Technology, Tsinghua University, Beijing, China.
His main research interests include flash-based storage systems and computational storage systems.

Keji Huang received the Ph.D. degree in electronic circuit systems from the University of Electronic Science and Technology, Chengdu, China, in 2004.
He is currently the Chief Architect of Huawei’s data storage product line and responsible for the overall architecture planning and technological evolution of Huawei’s storage products. He has more than 18 years of experience in ICT and more than 15 years of research experience in the storage field.
He is in-charge of multiple Huawei storage products, as well as the planning and design of large-scale architectures. His research interests include emerging storage architecture, large-scale distributed systems, and database systems.