Revisiting Swapping in User-space with Lightweight Threading

Kan Zhong[†], Wenlin Cui[†], Youyou Lu[§], Quanzhang Liu[†],
Xiaodan Yan[†], Qizhao Yuan[†], Siwei Luo[†], and Keji Huang[†]
[†]Huawei Technologies Co., Ltd
Chengdu, China
[§]Department of Computer Science and Technology, Tsinghua University
Beijing, China

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 disks. However, the heavy I/O stack makes the traditional kernel-based swapping suffers from several critical performance issues.

In this paper, We redesign the swapping system and propose ***LightSwap***, an high-performance user-space swapping scheme that supports paging with both local SSDs and remote memories. First, to avoids kernel-involving, a novel page fault handling mechanism is proposed to handle page faults in user-space and further eliminates the heavy I/O stack with the help of user-space I/O drivers. Second, we co-design Lightswap with light weight thread (LWT) to improve system throughput and make it be transparent to user applications. Finally, we propose a try-catch framework in Lightswap to deal with paging errors which are exacerbated by the scaling in process technology.

We implement Lightswap in our production-level system and evaluate it with YCSB workloads running on memcached. Results show that Ligthswap reduces the page faults handling latency by 3–5 times, and improves the throughput of memcached by more than 40% compared with the stat-of-art swapping systems.

CCS Concepts: • **Software and its engineering → Operating systems**.

Keywords: user-space swapping, memory disaggregation, light weight thread

1 Introduction

Memory-intensive applications[1, 2], such as in-memory databases, caching systems, 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, 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 [3–13] and still remains great challenges. Especially, we find that the heavy kernel I/O stack introduces large performance penalty, more than 40% of the time is cost by the I/O stack when swapping in/out a signal page, and this number will keep increasing if ultra-low latency storage media, such as Intel Optane [14] and KIOXIA XL-FLash [15] are adopted as the backend stores.

To avoid the high-latency of paging with SSDs, memory disaggregation architecture [16–20] proposes to expose a global memory address space to all machines to improve memory utilization and avoid memory over-provisioning. However, existing memory disaggregation proposals require new hardware supports [17, 18, 21–23], making these solutions infeasible and expensive in real production environments. Fortunately, recent researches, 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 several critical issues of these existing approaches.

First, kernel-based swapping, which relies on the heavy kernel I/O stack exhibits large software stack overheads, making it cannot 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 the kernel swapping, can be as high as 40μs even using one-side RDMA. Further, modern applications exhibit diverse memory access patterns, kernel-based swapping fails to provide enough flexibility to customize the eviction and prefetching policy. Second, recent research [25] has also explored the user-space swapping with userfaultfd, which however needs extra memory copy operations and exhibits ultra-high page fault latency under high-concurrency (i.e., 107μs under 64 threads), leading to systems that based on userfaultfd cannot tolerate frequent page faults. Finally, new proposals like AIFM [26] and Semeru [27], that do not rely on virtual memory abstraction, provide a runtime-managed swapping to applications and can largely reduce the I/O amplification. However, these schemes break the compatibility and require large efforts to rewrite existing applications.

Therefore, we argue that swapping need to be redesigned to become high-performance and remains transparent to applications. In this paper, we propose *Lightswap*, an user-space swapping that supports paging with both SSDs and remote memories. First, Lightswap handles page faults in user-space and utilizes the high-performance user I/O stack to eliminate the software stack overheads. To this end, we propose an ultra-low latency page fault handling mechanism to handle page faults in user-space (§4.2). Second, when page faults happen, existing swapping schemes will block the faulting thread to wait for data fetch, which lowers the system throughput. In Lightswap, we co-design swapping with light weight thread (LWT) to achieve both high-performance and application-transparent (§4.3). When page fault happens, leveraging the low context switch cost of LWT, we use a dedicated swap-in LWT to handle page fault and allow other LWTs to occupy the CPU while waiting for data fetch. Finally, due to the scaling in process technology and the ever increasing capacity, we find that both DRAM and SSD become more prone to errors. Therefore, we propose a try-catch exception framework in Lightswap to deal with paging errors (§4.4).

We evaluate Lightswap with memcached workloads. Evaluation results show that Lightswap can reduce the page fault handling latency by 3–5 times and improve the throughput by more than 40% when compared to Infiniswap. In summary, we make the following contributions:

- We propose an user-space page fault handling mechanism that can achieve ultra-low page fault notification latency (i.e., 2.4μs under 64 threads).
- We demonstrate that with the help of LWT, user-space swapping system can achieve both high-performance and application-transparent.
- We propose a try-catch based exception framework to handle paging errors. To best of our knowledge, we are the first work to explore handling both memory errors and storage device errors in a swapping system.
- We show the performance benefits of Lightswap with YSCB workloads on memcached, and compare it to other swapping schemes.

The rest of the paper is organized as follows. Section 2 presents the background and motivation. Section 3 and 4 discuss our design considerations and the design details of Lightswap, respectively. Section 5 presents the implementation and evaluation of Lightswap. We cover the related work in Section 6 and conclude the paper in Section 7.

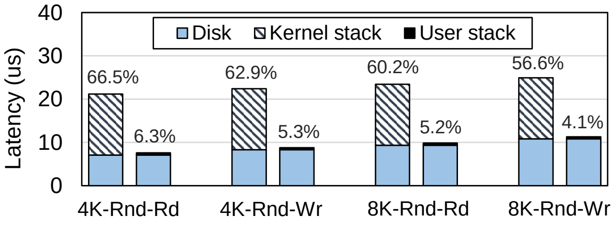


Figure 1. Random read and write latency breakdown. *The numbers on top of each bar denote the relative fraction of I/O stack time in the total latency.*

2 Background and Motivation

2.1 Swapping

Existing swapping approaches [3–10] depend on the existing kernel data path that is optimized for slow block devices, both reading and writing pages from/to the backend store would introduce high software stack overheads. Figure 1 compares the I/O latency breakdown for random read and write of XL-Flash [15] while using the kernel data path and user-space SPDK [28] driver. The figure shows that over half (56.6% – 66.5%) 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. Comparably, the I/O stack overhead is negligible while using the user-space SPDK driver.

Besides using local SSDs as the backend store, the high bandwidth and low latency RDMA network offers the opportunity for swapping pages to remote memories. The newly proposed memory disaggregation architecture [16–20] takes a first look on remote memory paging. In disaggregation model, computing nodes can be composed of large amount of memory borrowing space from remote memory servers. Existing works, such as Infiniswap [24] and FluidMem [25] have showed that swapping to remote memories is a promising solution for memory disaggregation. Nevertheless, Infniswap exposes remote memory as a local block device, paging in/out still needs to go through the whole kernel I/O stack. Our evaluation shows that the remote memory access in Infiniswap can as high as 40μs, which is about 10x higher than the latency of a 4KB page RDMA read. The difference is all caused by the slow kernel data path.

2.2 Linux eBPF

eBPF (for *extended Berkeley Packet Filter*) is a general virtual machine that running inside the Linux kernel. It provides an instruction set and an execution environment to run eBPF programs in kernel. Thus, user-space applications can instrument the kernel by eBPF programs without changing kernel source code or loading kernel modules. eBFP programs are written in a special assembly language. Figure 2 shows how eBPF works. As shown, eBPF bytecode can be loaded into the kernel using `bpf()` system call. Then a number of checks are performed on the eBPF bytecode by a verifier to ensure

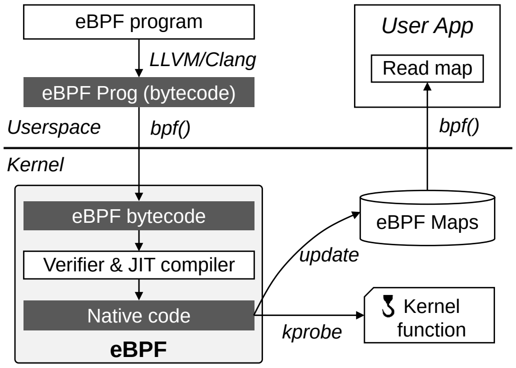


Figure 2. Linux eBPF framework.

that it cannot threaten the stability and security of the kernel. Finally, the eBPF bytecode can either be executed in the kernel by an interpreter or translated to native machine code using a Just-in-Time (JIT) compiler.

eBPF programs can be attached to predetermined hooks in the kernel, such as the traffic classifier (`tc`) [29] and eXpress Data Path (XDP) [30] in the network stack. One can also attach eBPF programs to `kprobe` tracepoint hooks, which makes eBPF programs can be attached to any kernel function. One of most important property of eBPF is that it provides *eBPF maps* for sharing data with user-space application. eBPF maps are data structures implemented in the kernel as key-value stores. Keys and values are treated as binary blobs, allowing to store user-defined data structures and types. To enable handling page fault in user-space, we utilize eBPF program to store thread context and page fault information into maps when page fault happens. Then our user-space page fault handler can retrieve the needed information from the maps to handle the page fault (§4.2).

2.3 Target Application and Execution Mode

In high-concurrency and low-latency memory-intensive applications, such as web service, in-memory cache and database, each server may handle thousands of requests. The traditional thread-based execution model (i.e., launching one thread per request) would lead to significant scheduling overhead, making most of the CPU time wasted on thread scheduling and context switching. To address this issue, light weight thread (LWT), as known as coroutine [31, 32] is proposed and widely adopted in memory-intensive applications to improve the throughput and reduce the response latency [33, 34]. Different from thread, such *pthread* in Linux, LWT is fully controlled and scheduled by user program, instead of the operating system. Each thread can be comprised of 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 much lower scheduling overhead and more flexible scheduling policy.

Moreover, in conventional swapping, when accessing a non-present page, the current thread will be blocked and woke up by the OS once the requested page is fetched back to local DRAM. During this process, the swap-in request needs to go through the whole IO stack to read the page, and a context switch is performed to put the blocked thread into running state, which both bring significant latency penalty.

Therefore, we adopt LWT as our application execution model for high-throughput in-memory systems, and co-design Lightswap with LWT to provide an high-performance and transparent user-space swapping to applications.

3 *LightSwap* Design Considerations

This section discusses the design considerations of Lightswap. To make Lightswap fast and flexible, we move page swapping from kernel to user-space, and co-design swapping with LWT to hide the context switching cost and improve the CPU utilization. Then, we discuss the need of paging error handling due to the scaling in process technology.

3.1 Why User space?

We design an user space swapping framework based on the following reasons:

1) User space I/O drivers show high potential in performance improvements. The performance of storage devices has been improved significantly due to the emerging technologies in both storage media and interface, such the Intel Optane memory [14], new NVMe (Non-Volatile Memory Express) [35] interface and PCI Express (PCIe) 4.0 interconnect. Therefore, the overhead of legacy kernel I/O stack becomes more and more noticeable since it was originally optimized for slow HDDs.

To reduce the I/O stack overhead, user space I/O stacks without any kernel intervention are desired to support high-performance storage device. To this end, Intel released storage performance development kit (SPDK) [28], which moves all the necessary drivers into user-space, thus avoids syscalls and enables zero-copy. Other optimizations, such as polling and lock-free are also used in SPDK to enhance the I/O performance. To accelerate network I/Os, DPDK [36], a packet process acceleration framework, maps Ethernet NIC into user-space and control it in user-space program. DPDK also provides an user-space polling mode driver (PMD), which uses polling to check for received data packets instead of using interrupts as the kernel network stack would. Therefore, to be beneficial from these high-performance user-space I/O drivers, we build Lightswap in user-space.

2) User space swapping can easily support memory disaggregation. Thanks to the fast and low-latency RDMA network, the effective memory capacity can also be extended through remote memory paging. To achieve this, memory disaggregation architecture [16–20] has been proposed to expose the memories in dedicated servers to computational severs to enlarge their memory space. Previous works [24, 25, 37, 38] have shown that swapping is a promising solution to enable efficient memory disaggregation.

Lightswap uses an user-space key-value abstraction for paging in/out. Pages are read/write from/to backend stores through an unified KV interface. Thus, memory disaggregation can be enabled by writing pages to a remote KV store in Lightswap.

3) User-space is more resilient to errors. Due to the continuous scale in storage density and process technology, both storage devices and memories becomes more prone to errors [39]. When these errors are triggered in kernel space, Linux and UNIX-like OSes have to call a panic function, which cause system crash and even service termination.

To deal with the above errors in user-space, one can isolate the faulted storage device or memory address and then simply kill the corresponding applications. However, this approach still causes application termination and thus lower the system's availability. Moreover, it is difficult to handle the error properly without application semantics. Therefore, we propose to handle memory and device errors in user-space with application-specific knowledges.

3.2 Swapping with LWT

Existing OS usually pays a high cost on thread context-switching, which can take 5–10 microseconds on x86 platforms [40]. To hide the thread context-switching latency, Fastswap [41] poll waits the requested page when page fault happens by leveraging the low latency of RDMA network. However, with SSD-based swapping or larger page size, reading the requested page into local memory needs comparably longer durations. Even paging with remote memory with RDMA, we still argue that polling wait is not the optimal way as the context-switching of LWT is in nanoseconds. Thus, polling wait would cause a large waste on CPU cycles. To tackle this issue, Lightswap uses asynchronous I/O: it switches to other LWTs while waiting for data fetch.

To effectively swap-in pages in user-space, we co-design Lightswap with LWT (§4.3). First, to make Lightswap be transparent to applications, Lightswap uses a dedicated LWT, referred as *swap-in LWT*, to fetch pages from a backend store to local memory. When a page fault is triggered by normal application LWT, referred as *faulting LWT*, it will be blocked and the swap-in LWT is launched to fetch the requested pages. Second, the swap-in LWT will also be blocked and yields CPU for other worker LWTs when waiting for data fetch. Finally, to reduce the overall page fault latency, we adjust our LWT scheduler to prioritize swap-in LWTs, thus the requested pages can be fetched as soon as possible.

3.3 Handling Paging Errors

Due to storage device and memory errors, pages in SSD-based backend store, remote memory, and local memory have more possibility to be corrupted. Therefore, paging mainly encounter two kinds of errors: 1) *swap-in error*, swapped out pages cannot be brought back due to device or network failure, and 2) *uncorrectable memory error (UCE)*, memory error that has exceeded the correction capability of DRAM hardware. Existing data protection methods, such as replication and erasure code, only work well for slow disk. To reduce the possibility of encounter memory errors in memory access, a daemon named `memory scrubber` will periodically scan DRAM and correct any potential errors. However, the most advanced DRAM ECC scheme (i.e., chipkill) also fails to correct errors from multiple devices in a DIMM module [42]. When an UCE is found by memory scrubber or triggered during memory access (i.e., load/store), the BIOS will generate a hardware interrupt to notify the OS that a memory UCE has happened. To deal with these paging errors, including swap-in errors and UCEs, 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 also lower the system's availability.

Fortunately, some applications, such as in-memory caching system, can tolerate such memory data corruption/loss as they can recovery data from disk or replicas. Therefore, in Lightswap, we propose an error handling framework, which provides an opportunity for applications to tolerate and correct these errors in the application context. When a paging error happens, the corresponding application is notified and then the application will try handling this error using its specific error handling routine.

4 *Lightswap* Design

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

4.1 Lightswap Overview

Figure 3 illustrates the overall architecture of Lightswap. As shown, Lightswap handles page faults in user-space and uses a generic key-value store for swapping in/out. For the key-value store, keys are the process virtual addresses, while values are pages. Using key-value interface for swapping makes pages can be swapped to arbitrary storage devices. With one-side RDMA semantics, memory disaggregation can also be enabled by swapping pages to remote memory pools. The components of Lightswap are introduced as below.

LWT library. An application can create multiple standard pthreads, which usually are bounded to given CPU cores. The number of pthreads is limited by the number of available CPU cores to minimize scheduling overhead. Inside each pthread, LWTs are created to process user or client requests. LWT library is provided to user application for creating and managing LWTs. In the LWT library, a scheduler is designed for scheduling LWTs based on the priority. Different from

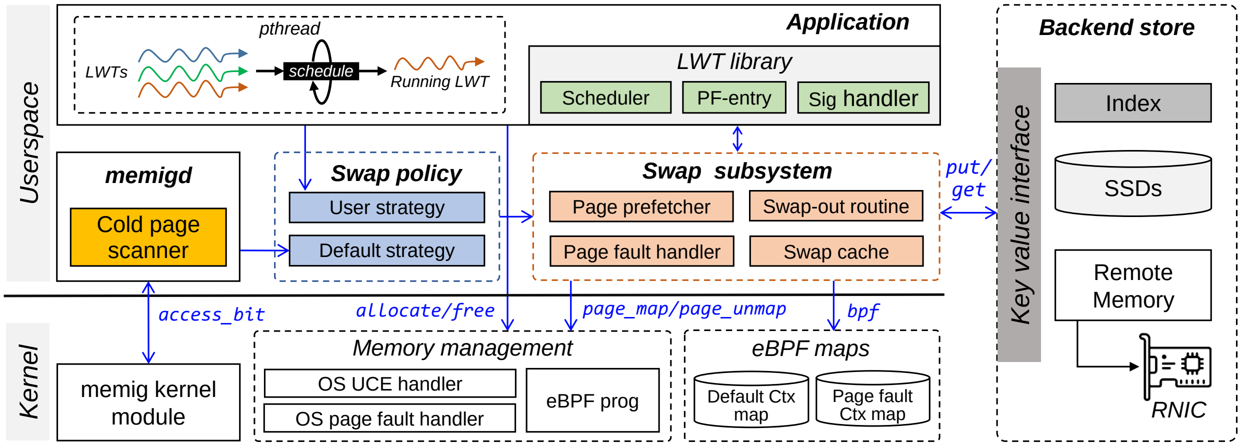


Figure 3. Lightswap architecture. Lightswap handles page faults in user-space and swap in/out pages uses a key-value interface.

thread scheduling and context-switching, which requires kernel involvement, the scheduling of LWTs is fully controlled by LWT library in user-space without any kernel efforts. Therefore, the scheduling overhead of LWT is minimized. In our measurement, LWT switching latency is usually less than 1 microsecond, which is orders of magnitude faster than thread switching (several microsecond).

To handle page faults triggered by LWTs, each pthread has a page fault entry point (`PF-entry`), which is the entry point of user-space page fault handler. When a page fault happens, this entry point will be reached and then the user-space page fault handler will be involved after reading the page fault information from eBPF maps (§4.2).

The signal handler (`Sig handler`) is used to receive paging error signals. For memory UCEs, the BIOS will notify the OS to handle the error, and the OS UCE handler will first tries to isolate the faulted memory address and then issues a signal to the user application, more specifically, to the `Sig handler`. For swap-in errors, signals will be generated and sent by the swap-in LWTs. In the signal handler, application-specific paging error handling routine will be executed to try to resolve the error (§4.4).

Memory migration daemon (memigd). `memigd` is a user-space process that responsible for scanning cold pages of given applications. The results will be used as the input of default strategy in swap policy to guide the page reclaiming. To identify cold pages accurately, `memigd` utilizes a kernel module, namely `memig` to periodically test and clear the access bit of page table entries. The access bit of a page table entry is set to '1' by hardware once the page is touched. Therefore, in `memigd`, pages whose access bits are survived during two consecutive scans are considered as hot pages, otherwise, they will be regarded as cold pages. When the system's available memory below a pre-defined threshold, `memigd` will start to scan cold pages of user applications that labeled as swappable, then these cold pages will be selected as victim pages for swapping out. In Lightswap, to reduce the scanning cost and make mission-critical applications fully reside in memory, only applications that marked as swappable will be scanned for paging out. Moreover, to reduce the I/O operations, we also only write dirty victim pages back to backend store, clean victim pages are discarded directly as they already existed in the backend store.

Besides cold page identification, `memigd` can also be employed to control the physical memory usage of each process, enabling memory quota for user applications. Once the system's memory is under pressure or applications' memory usage exceeds the quota, `memigd` will starts scan cold pages and notify the `uswplib` (see below) for swapping out.

User-space swap library (lswaplib). `uswplib` is the core component of Lightswap. It is a library that enables user-space swapping for applications. `uswplib` is comprised of two parts: swap policy and swap subsystem. The swap policy decides which pages to be swap out based on both user specific strategy and default strategy. Besides, swap policy can also be used to guide the page pre-fetching when page fault happens. More specifically, applications can pass application-specific strategy to swap policy by calling:

```
void swap_advise(void* addr, size_t len, bool out).
```

If parameter `out` is true, this function provides swap out suggestions. Pages in *(addr, len)* of that application will be preferred for swapping out. In our current implementation, if application memory quota is configured, pages belong to *(addr, len)* will be swapped out once the application's memory quota is full, otherwise, the swap policy will evenly select victim pages (including cold pages recognized by `memigd` and pages suggested using `swap_advice()`) across applications for swapping out.

If parameter *out* is false, this function provides prefetch hints. Since we do not know when these pages will be used, to reduce the memory footprints, we neither bring in pages

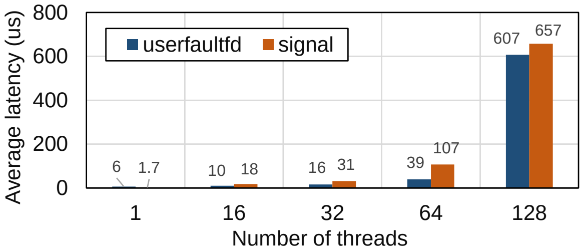


Figure 4. Average page fault events notification latency under different number of threads.

in *(addr, len)* immediately nor bring these pages in a consequent page fault handling process. Instead, we bring in pages belong to *(addr, len)* only when an address in this range causes a page fault. In the corresponding page fault handling routine, we prefetch these pages from the backend store.

The swap subsystem do the actual paging in and out. A dedicated swap-out thread is launched to receive victim pages from swap policy and write them to backend store. When swapping pages out, pages will be first removed from application's page table use `page_unmap()` system call and then added to the swap cache. Pages in swap cache are asynchronously writeback to the backend store, thus decoupling page write from the critical path of swapping out routine. A dedicate thread periodically flushes the swap cache to the backend store when its size has reached a pre-configured threshold batch size. `uswplib` defines an user-space page fault handler, which will be called to swap in the requested page when page fault happens. The page fault handler will first search the swap cache for the desired page. If the page is found in the swap cache, we remove it from the swap cache and add it to the application's page table at the faulted address. Otherwise, page will be read from the bankend store. In the page fault handler, we also decouple the prefetching from the critical path of swapping in routine. After bringing the desired page into memory, the page fault handler appends a prefetch request into a prefetch queue. If the faulted address associated with an application-specific prefetch hint, the prefetch request will read pages in *(addr, len)* specified by `swap_advice()`. Otherwise, a simple read-ahead policy will be used and the prefetch request tries to read the surrounding eight pages at the faulted address. To improve the concurrency and of page prefetching, a group of I/O LWTs, referred as *prefetchers* will constantly pull requests from the prefetch queue and bring pages into memory. Note that in our current implementation, we do not have any special or carefully designed prefetch algorithm as this work does not aim at prefetching, but these algorithm can be easily added to Lightswap.

eBPF prog. The `eBPF prog` is eBPF bytecode that injected into the kernel using `bpf()` system call. It is the key component of handling page fault in user-space (§4.2). eBPF prog maintains two context maps: default context map and page fault context map. Both maps contains multiple entries. Each entry of the default context map stores the default context that is saved at page fault entry point (i.e., `PF-entry`). Each entry of the page fault context map stores the LWT context where page fault is triggered. When page fault happens, `eBPF prog` is responsible for 1) saving the context of faulting LWT into the page fault context map and 2) modify the current context with the previously saved default context.

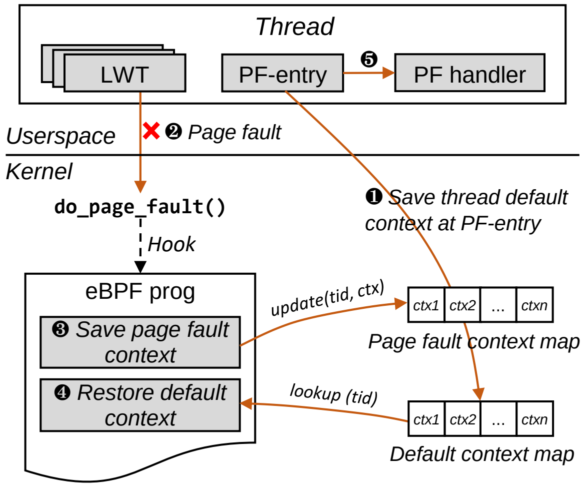


Figure 5. eBPF-based page fault handling scheme.

4.2 Handling Page Fault in User-space

Page faults are handled in user-space in Lightswap. The key challenge here is how to notify user-space application when page fault happens effectively. To achieve this, Linux provides userfaultfd [43] to notify and allow user-space applications handle page faults of pre-registered virtual address regions. Besides, one can also use Linux signal to notify user-space applications that page faults happened, which is similar to the memory UCE notification (§4.4). However, both userfaultfd and signal suffer from several performance issues as discussed below.

Userfaultfd requires a fault-handling thread that pulls page fault events through reading the userfaultfd file descriptor, and provides `UFFD_COPY` and `UFFD_ZERO` ioctl operations to resolve the page faults. When a page fault happens, The OS page fault handler puts the faulting thread into sleep and allocates physical page for the faulted address, then an event is generated and sent to the fault-handling thread. The fault-handling thread reads events from userfaultfd file descriptor and resolves the page fault with `UFFD_COPY` or `UFFD_ZERO` ioctl operations. In userfaultfd, all the page faults are handled by the fault-handling thread, which will easily become the bottleneck when multiple threads trigger page faults almost simultaneously. Moreover, one cannot directly bring the request page from backend store to the faulted address. To resolve a page fault, one must first read the request page from backend store to a memory buffer, and then copy the

```
pthread_func() {
   ucontext *ctx = get_current_contex();
   PF-entry(ctx);
   while (current->stat != EXITING)
      LWT_sched();
}
```

```
 PF-entry(ucontext *ctx) {
    if (ctx_saved == FALSE) {
       ebpf_map_save_default_ctx(ctx);
       ctx_saved = TRUE;
       return;
    }
    ucontext *pf_ctx;
    pf_ctx = ebpf_retrieve_pagefault_ctx();
    pagefault_handler(pf_ctx);
 }
```

Figure 6. Swapping with LWT. Thread schedules LWTs and calls page fault handler when any LWT triggers page fault.

page from the buffer to the faulted address using `ioctl` system call, which brings one extra memory copy operation.

For the signal approach, when page faults happen, the OS page fault 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 makes the signal sending routine suffers from seriously lock contention under high-concurrency.

To show the performance of userfaultfd and signal, we record the page fault notification latency (i.e., latency from page fault happens to the user-space page fault handler receives the page fault event) of both userfualtfd and signal under different concurrent threads and plot their average latency in Figure 4. The detailed configurations can be found in §5.3. As shown, with only one thread, both userfaultfd and signal achieve very low page fault notification latency (i.e., 6μs for userfaultfd and 1.7μs for signal). However, with the increasing of concurrent threads, the page fault latency of both userfaultfd and signal increase significantly. Therefore, neither userfaultfd nor signal is impractical for handling page fault in user-space for high-currency applications.

To effectively handling page faults in user-space, we propose the eBPF-based page fault notification scheme. As shown in Figure 5. In the main thread, before launching and scheduling LWTs, thread will enter the page fault entry point (i.e., `PF-entry`) and saves the current thread context into the default context map (❶). When one of the LWT triggers a page fault (❷), the kernel page fault handler will be involved. We use the kernel page fault handling function (i.e., `do_page_fault()`) as a hook point and attached our eBPF program to this function. Once this function is involved, the attached eBPF program will be executed. In the eBPF program, we first save LWT's context at the point that page fault happens (❸). We refer this context as page fault context and store it into the page fault context map, which uses the thread ID (i.e., `tid`) as the key and the thread context as its value. The page fault context contains the page fault address and will be used to restore the execution of LWT after page fault 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 context is modified to the retrieved default context, which makes the execution of current thread restore to `PF-entry` (❹). In `PF-entry`, thread will notice that the default context is already saved, which means this is not the first enter and thus thread knows page fault occurs. Then, the page fault context is also retrieved from the page fault context map and saved in the faulting LWT's stack. The page fault context will be employed by the LWT scheduler to restore the execution of the faulting LWT. Finally, the user-space page fault handler is called to resolve the page fault (❺). In the user-space page fault handler, the faulting LWT is blocked and put into sleep, then the requested page will be read from the backend store (see details in §4.3). Once the page fault is resolved, the state of the faulting LWT is set to runnable and will be scheduled in the next scheduling period. In the next subsection, we will show how pages are read from backend store using LWT.

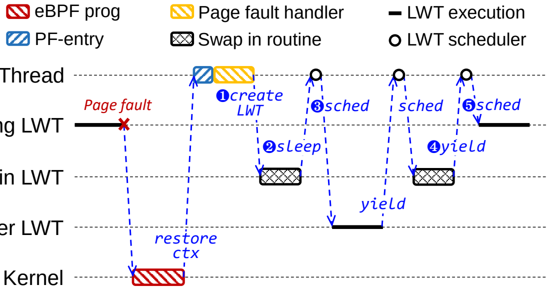


Figure 7. Scheduling of swap-in LWTs and faulting LWTs.

4.3 Co-design Swapping with LWT

Lightswap handles page faults in user-space. There are two challenges that we need to address. First, user applications must use Lightswap transparently to avoid application modifications. Second, to reduce the total page fault latency, the faulting LWT must be woken up as soon as possible after the requested page is brought into memory.

To address these issues, we co-design swapping with LWT to reduce the swap-in latency. Figure 6 shows the pseudo code of how LWTs are scheduled and user-space page fault handler are called. In the main thread, the thread's context is saved to the default context eBPF map in `PF-entry()` before scheduling LWT. After that, the LWT scheduler (i.e., `LWT_sched()`) continuously picks and run LWTs from the front of the ready LWT queue. When any LWT triggers page fault, the faulting LWT is blocked and the thread is restore to `PF-entry()`, in which the user-space page fault handler will

```
1: LIGHTSWAP_TRY {
2:     // do something
3:     *addr = value;
4:     // do something
5: } LIGHTSWAP_CATCH (_err_code, _vaddr) {
6:     /* paging error handling code */
7: } LIGHTSWAP_TRY_CATCH_END
```

(a)

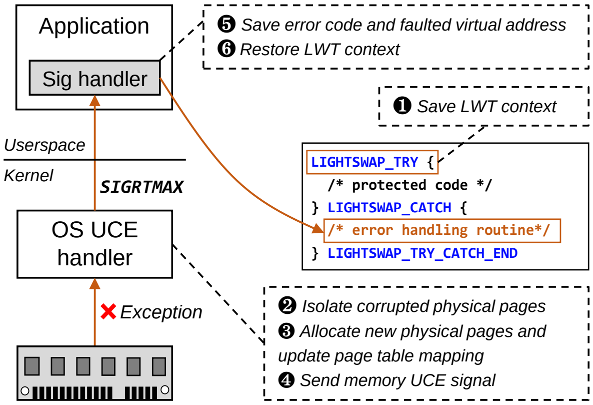
```
1:  #define LIGHTSWAP_TRY
2:  do {  \
3:     ucontext *lwt_ctx = getcontext(); \
4:     if (lightswap_has_err()) {
5:
6:  #define LIGHTSWAP_CATCH (_err_code, _vaddr)
7:     } else {  \
8:         int _err_code = get_error_code(); \
9:         ulong _vaddr  = get_pagefault_addr();
10:
11: #define LIGHTSWAP_TRY_CATCH_END
12:    }  \
13: while(0);
```

(b)

Figure 8. Try-catch paging error handling framework. *(a) Example of how application handling paging errors with the proposed try-catch framework; (b) Lightswap try-catch keywords macro definition.*

be called. To tackle the first challenge, a new LWT (referred as *swap-in LWT*) is created to swap in the requested page in the page fault handler. Thus, user application does not aware page fault happens and the page fault will be handled by our dedicated swap-in LWT. To tackle the second challenge, we make the LWT scheduler prefers swap-in LWTs and faulting LWTs. To achieve this, we classify the ready LWTs into three queues: 1) *swap-in LWT queue*, swap-in LWTs are put into this queue after they are created and ready to run; 2) *faulting LWT queue*, which contains LWTs that encounter page faults and the page faults have been resolved, which means their requested pages have been brought into memory by the swap-in LWTs and their status become ready; 3) *normal LWT queue*, other ready LWTs are resided in this queue. The LWT scheduler assigns the first priority to the swap-in LWT queue, second priority to the faulting LWT queue, and third priority to the normal LWT queue. Therefore, swap-in LWTs can be scheduled to run immediately after they are created, and once the requested pages are swapped back to memory, the faulting LWTs can be scheduled to run as soon as possible.

Figure 7 illustrates an example of scheduling of swap-in LWT that reading page from backend store. As shown, when page fault happens, the faulting LWT is blocked and a dedicated swap-in LWT is created in the user-space page fault handler (❶). We assume that the swap-in LWT queue is empty currently, thus when the swap-in LWT is added to the swap-in LWT queue, it will be scheduled to run immediately.


Application
Sig handler
❺ Save error code and faulted virtual address
❻ Restore LWT context
❶ Save LWT context
Userspace
Kernel
SIGRTMAX
OS UCE handler
Exception
❷ Isolate corrupted physical pages
❸ Allocate new physical pages and update page table mapping
❹ Send memory UCE signal

```
LIGHTSWAP_TRY {
  /* protected code */
} LIGHTSWAP_CATCH {
/* error handling routine*/
} LIGHTSWAP_TRY_CATCH_END
```

Figure 9. Handling memory UCE.

After the requested page is brought into memory, the swap-in LWT changes the state of faulting LWT to ready and adds it to the faulting LWT queue. Then it gives up the CPU by calling `yield()` (❹). Finally, the scheduler picks the faulting LWT from the queue and schedules it to run (❺). To improve the throughput, we propose swap in pages asynchronously by leveraging the negligible scheduling overhead of LWT. As shown in Figure 7, the swap-in LWT is put into sleep when waiting for page to be read from backend store (❷). Thus, other ready LWT can be scheduled to run to maximize the CPU usage(❸). After the page is brought into memory, the swap-in LWT is set to ready and re-added to the swap-in LWT queue. Remember that swap-in LWT queue has the highest priority, and thus the swap-in LWTs will be preferred by the scheduler at the next LWT scheduling.

4.4 Try-catch Exception Framework

Inspired by the try-catch exception handling approach in C++, we designed a paging error handling framework in Lightswap. Basically, applications can embed `LIGHTSWAP_TRY` and `LIGHTSWAP_CATCH` macro into their program, as shown in Figure 8(a). Codes that surrounded by `LIGHTSWAP_TRY` macro will be protected against from paging errors. For the example in Figure 8(a), if the pointer deference at line 3 triggers a paging error (memory UCE or swap-in error), the application will jump to the `LIGHTSWAP_CATCH` immediately to handle the paging error using application customize codes. For example, memory cache applications can recovery the data from disk. Through this way, we provide an opportunity for applications to handle paging errors in user-space.

Figure 8(b) shows the definition of Lightswap try-catch macro. In `LIGHTSWAP_TRY` macro (line 1–4), we first save the context of the current LWT (line 3) and check whether a paging error is encountered (line 5). In normal execution, `lightswap_has_err()` returns false and codes in the `LIGHTSWAP_TRY` block will be executed. If a paging error happens when executing codes in the `LIGHTSWAP_TRY` block, the user-space page fault handler or the signal handler will restore the program pointer to the context saving point (line

Number of threads/LWTs		1	16	32	64	128
Latency (μs)	Userfaultfd	6	10	16	39	607
	Signal	1.7	18	31	107	657
	eBPF	1.6	1.7	2	2.4	4

Table 1. Average latency of different user-space page fault notification schemes.

3) using the previous saved LWT context. Then, function `lightswap_has_err()` will return true as an error happens, which makes the application jumps to `LIGHTSWAP_CATCH` block to handle the error.

Figure 9 shows how memory UCE is handled in Lightswap. For memory access that causes memory UCE, the OS UCE handler will first be notified by the hardware. In the OS UCE handler, the corrupted physical pages are isolated and new pages are allocated and mapped to the faulted virtual addresses (❶❷). Then a standard Linux signal, which contains the signal number, error type (i.e., memory UCE), and faulted virtual address is sent to the corresponding application (❸). Currently, we utilize the maximum signal number (i.e., `SIGRTMAX`) as the memory UCE signal. After the signal handler captures the signal, it first saves the error code and faulted virtual address, which will be used in the `LIGHTSWAP_CATCH` block, then it restore LWT context to the point that the context is saved (❹❺). With these steps, the application will finally jump to the `LIGHTSWAP_CATCH` block to handle the memory UCE.

To handle swap-in errors, the user-space page fault handler is responsible for restoring the LWT context. For example, if the pointer deference at line 3 in Figure 8(b) triggers a page fault and the user-space page fault handler finds that the requested page cannot be brought in memory correctly, it first saves the error code and faulted virtual address, and then restores the LWT context to let application to handle the swap-in error.

5 Evaluation

This section introduce the evaluation of Lightswap, we start with a brief introduction of our system implementation, then give the evaluation setups and finally discuss the results.

5.1 Implementations

We implement and evaluate Lightswap in real production system to show its effectiveness. We made some modifications to the Linux kernel to let it supports user-space swapping effectively. First, we add a hook point, more especially, a empty function to the kernel. The kernel page fault handler (i.e., `do_page_fault()`) will call this function and then return immediately if the faulted address belongs to an application that is supported by Lightswap. We make the eBPF program to hook this function and thus normal page faults still use the kernel page fault handler while only applications that are supported by Lightswap use our user-space page fault handler. To reduce the number of `bpf()` system calls, we use shared memory between kernel and user-space to share the page fault context map. Second, to support swap in/out pages in 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, thus the swap-in LWT can update the page table mapping for the faulted address, and the swap-out thread can also remove a page from the application's page table. Third, we modified the OS UCE handler to make it send memory UCE signal to our signal handler if the OS cannot correct the memory error. Totally, all these kernel modification effort is no more than 1000 lines of code.

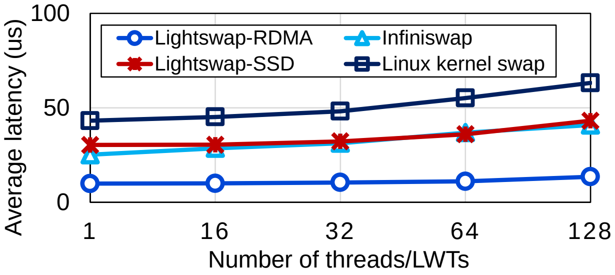


Figure 10. Average page fault handling latency, from page fault happens to the requested page be brought into memory by the page fault handler. *Lightswap-RDMA denotes paging with remote memory via one-side RDMA, while Lightswap-SSD represents paging with local SSDs.*

To implement the key-value based backend store, we use a in-memory hash table as its index to reduce the index traversal time. We use a second hash table to solve the hash conflicts. Entries in the second hash table point to the actual positions of pages. For local SSDs, we organize the SSD space in a log-structured way and thus pages in 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 1GB large memory blocks and response clients their registered memory region IDs and offsets for RDMA. The backend store in the client is responsible for managing memory blocks and splitting them into pages.

5.2 Evaluation Setups

We employ two x86 servers in the evaluation, one is used as client for running applications. Another server is configured as memory server to allocate memory blocks. Each server equips with two Intel Xeon CPUs, and each CPU contains 40 cores with hyper-thread enabled. The memory capacity of both server is 256GB. We will limit the memory usage of client server in order to trigger swapping in/out. The connection between two serves is 100G RoCE with our customized user-space driver that based on DPDK. For paging with local SSDs, we use the state-of-art NVMe SSD with our SPDK-based user-space NVMe driver as the storage device.

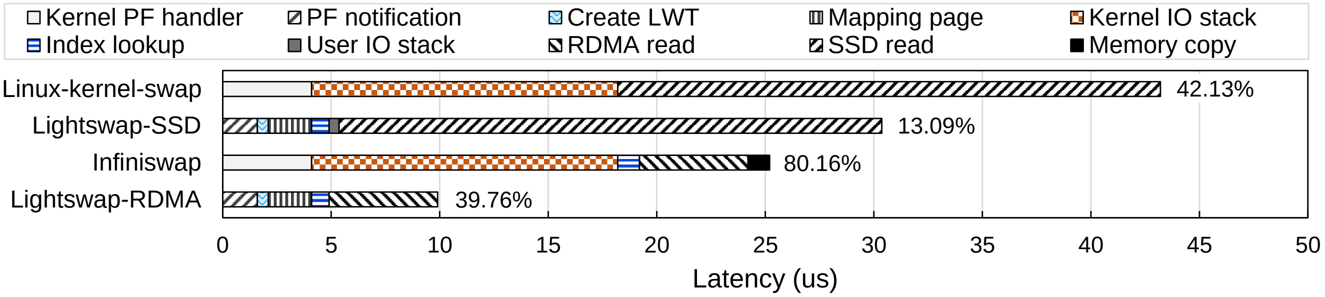


Figure 11. Page fault handling latency breakdown under no concurrency. *The numbers beside each bar denote the fraction of time cost by the software stack during handling page faults.*

5.3 Microbenchmarks

Page fault notification latency. To show the effectiveness of eBPF-based user-space page fault handling, we first evaluate the average page fault notification latency under different concurrency, where the page fault notification latency denotes the latency from page fault happens to the user-space page fault handler receives the page fault event. We compares the average page fault notification latency among eBPF, userfaultfd and signal. Since our eBPF-based approach is co-designed with LWT, to evaluate its performance, we create multiple threads (from 1 to 128) and bound them to certain CPU cores. Inside each thread, we launch a LWT as the faulting LWT. For userfaultfd, we use one faulting-handling thread to pull page fault event and create multiple threads as faulting threads. For the signal approach, a signal handler is registered as the user-space page fault handler. In this scheme, we reuse the signal number of handling swap-in errors and memory UCEs, and use error code to identify the actual fault type (i.e., page fault, swap-in error or memory UCE). In the page fault handler of all these schemes, we simply allocate and map a zeroed page for the faulted address and return immediately.

Table 1 shows the average page fault notification latency of handling page faults in user-space. As shown, when there is no concurrency, all of these page fault notification schemes perform well, achieving extreme low latency. However, with the increase of concurrency, the notification latency of both userfaultfd and signal increase exponentially. With 128 threads, the average latency of userfaultfd and signal as high as 607μs and 657μs, respectively. As we discussed in §4.2, for userfaultfd, the high latency is caused by the contention of fault handling thread. The fault handling thread can launch multiple threads to handle page faults concurrently, but this also brings extra CPU overheads and adds synchronous costs. For the signal scheme, in the signal sending routine (i.e., `force_sig_info()`), a `siglock` must be obtained before sending the signal, which leads to seriously lock contention under high-concurrency and thus resulting the high latency of page fault notification. In the contrary, the proposed eBPF-based page fault notification scheme achieve extreme low latency under all the degrees of concurrency. When the number of LWTs increases from 1 to 128, the average latency only has a slight increment, which is mainly due to the lock contention of eBPF maps. In our current implementation, we employ 32 eBPF hash maps for page fault context, and a lock is used to protect each map. We divide the page fault contexts of LWTs into these eBPF maps evenly by using the core ID as an index number. Thus, even with 128 LWTs, each eBPF map only needs to store the page fault contexts of 4 LWTs, which significantly reduces the lock contention and contributes to the slight increment of latency under high-concurrency.

Page fault handling latency. To show the end-to-end performance of Lightswap, we compare the page fault handling latency of Lightswap to other swapping schemes. We denotes the page handling latency as the time duration from page fault happens to the page fault handler finishes resolving the page fault. We compare the results between Lightswap, Infiniswap and the Linux default kernel swap. Figure 10 illustrates our evaluation results. Note that page fault handling latency does not include the time duration from the page fault be resolved to the point that the faulting thread/LWT is restored to run. As shown, when paging with remote memory through one-side RDMA, Lightswap achieves the lowest page fault handling latency, ranges from 10μs to 13.5μs under different degrees of concurrency. The conventional Linux kernel swap has the highest page fault handling latency, ranges from 43.2μs to 63.2μs. 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 page fault handling latency. Even paging with local SSDs, the proposed Lightswap achieves comparable performance with Infiniswap, and has around 30% lower latency than Linux kernel swap. With ultra-low latency NVMe SSDs, such as Intel Optane and KIOXIA XL-Flash, we believe that Lightswap-SSD can achieve lower latency than Infiniswap.

To demystify the reason behind this improvements, we breakdown the page fault handling process and plot its detailed time cost in Figure 11. In the figure, the kernel PF (page fault) handler has already included the time spend on trap into the kernel. In our measurement, reading a 4KB page

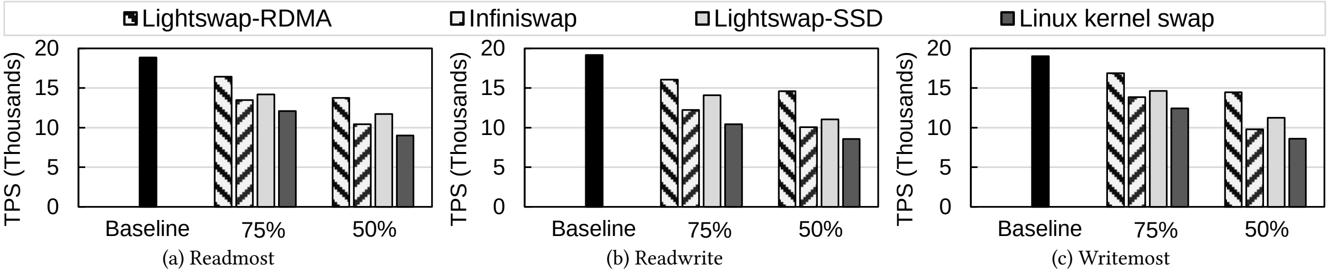


Figure 12. Average TPS of memcached with different swapping schemes. *We compare the performance of different swapping schemes with 75% and 50% physical memory of the memcached dataset size. We use 32 worker threads to process requests for all these configurations.*

from remote memory via one-side RDMA and local SSD will respectively cost 5μs and 25μs on average in our environment. Lightswap handles page faults in user-space and avoids the slow kernel data path by leverage high-performance user-space drivers. Therefore, in Lightswap, the page fault handling latency is dominated by the page read latency. Software stack respectively takes 39.76% and 13.09% for paging with remote memory and local SSDs. However, both Infiniswap and Linux kernel swap need to go through the entire kernel I/O stack when fetch pages, making the kernel I/O stack takes a large fraction of the total latency. The kernel I/O stack is mainly comprised by the generic block layer that provides OS-level block interface and I/O scheduling, and the device driver that handles device specific I/O command submission and completion. As shown in the figure, due to the kernel I/O stack, the software stack overheads for Infiniswap and Linux kernel swap are 80.16% and 42.13%, respectively. Despite the fact that Infiniswap also one-side RDMA, the high software stack overhead makes its page fault handling latency reaches 25μs, and even exceeds 40μs under high-concurrency.

5.4 Application: Memcached

Memcached is an widely used in-memory key-value based object caching system. Memcached uses the client-server mode and in the server sides, multiple worker threads is created to process the PUT and GET requests from the client side. We benchmark memcached with the YCSB workloads [44] under different swapping schemes. Each YCSB workload performs 1 million operations on memcached with 10,485,760 1KB records, which are 10GB data in total. Table 2 summarizes the characteristics of our YSCB workloads.

Workload Name	Read	Insert	Update	OPs	Size
Readmost	90%	5%	5%	1 million	10GB
Readwrite	50%	25%	25%	1 million	10GB
Writemost	90%	5%	5%	1 million	10GB

Table 2. YCSB workloads characteristics.

In memcached, mutiple worker threads are created to handle requests from the client-side. However, the proposed Lightswap is co-designed with LWT, so we first rewrite memcached and make it to use LWTs to process client-side requests. In this LWT version memcached, we create mutiple worker threads and bound these threads to certain CPU cores. Inside each thread, we launch a worker LWT for each incoming request. We limit the number of LWTs in each thread to 10 to reduce the LWT management overhead. Since the LWT execution mode is much similar to the thread mode, the rewritting does not need too much efforts. In our system's LWT implementation, each LWT has a maximum 32KB stack, and the execution of LWTs is non-preemptive, which means they will hold the CPU till get terminated or reach a waiting state (e.g., waiting semaphore) or proactively release the CPU by calling `yield()`. For other swapping schemes, we still use the thread version memcached.

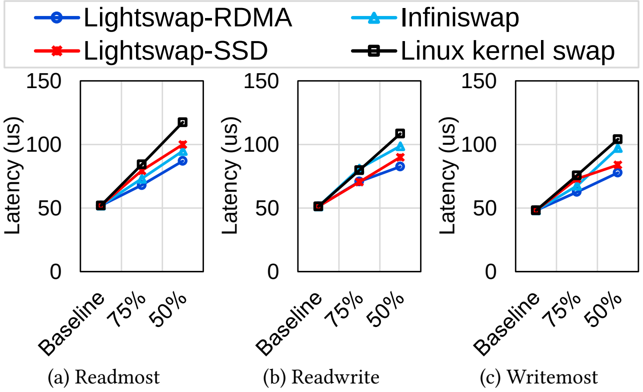


Figure 13. Comparison of average operation latency under different swapping schemes.

Figure 12 compares the average throughput of different swapping schemes. Besides, we also compares their average operation latency in Figure 13. In these two figures, The baseline indicates the case that all memcached's dataset is reside in memory and there is no swap-ins/-outs. Since the number of threads/LWTs in memcached is smaller than the number CPU cores (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. We found 3 key results

Error type	Count	Terminated (%)	Survived (%)
Swap-in error	10000	43%	57%
Memory UCE	15000	74%	26%

Table 3. Paging error handling results. *We use the OS UCE handler and the swap-int LWT to randomly generate memory UCEs and swap-in errors, respectively.*

from these figures: 1) Lightswap-RMDA has the least performance degradation, it outperforms Infiniswap and Linux kernel swap by 40% and 60% on average in throughput, respectively; 2) Due to the outstanding page fault handling latency, Lightswap-RDMA also achieves the lowest latency among these swapping schemes, it outperforms Infiniswap and Linux kernel swap by 18% and 30% on average, respectively; 3) Even with higher operation latency, Lightswap-SSD still achieves 10% – 20% higher throughput than Ininifswap. This is mainly because that Lightswap is co-desinged with LWT. In LWT version memcached, page fault insteads of blocking the current worker thread, it only blocks the faulting LWT. Thus other worker LWT can still get scheduled and executed by the worker thread. In contrast, in the thread version memcached, the current worker thread will be blocked once page fault happens, leading to that the CPU usage cannot be maximized.

In order to show the effectiveness of the propose paging error handling framework, we generate random UCEs and swap-in errors for address space used by memcached. Currently, we only add a simple paging error handling routine in `do_item_get()` of memcached. Since the item metadata and item data are placed in the same structure, and items belong to the same slab class are linked in the same LRU list. Thus, if the error handling routine finds that any item is corrupted due to paging error, it has to reset the whole slab class and returns not found for GET requests. If paging error causes any corruption in the memcached metadata, such as the hash table and slab class array, the error handling routine has to terminated memcached.

Table 3 shows the results of simulated paging error handling results for Readmost workload, we generate 10 thousands swap-in errors and 15 thousands memory UCEs to memcached. As we only protect the GET operation, most of the memory UCE will cause process termination, memcached only survives in 26% of the errors. In contrast, memcached survives in most of case (i.e., 57%) of swap-in errors as the test workload is dominated by GET operations. We believe that with more try-catch protections, memcached can eliminate more process terminations.

6 Related Work

SSD-based swapping. Swapping has been studied for years, with magnitudes of performance 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 [3–10] to enlarge the main memory. They are integrated with Linux virtual memory and rely on 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 [11–13] are proposed to fully exploit flash's performance 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 traffics need to go through all the 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 [45–51] 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 hardwares, it has became possible to reorganize resources into disaggregated clusters [16–20, 52–54] to improve and balance the resource utilization. To achieve memory disaggregation, Fastswap [41] and INFINISWAP [24] explore paging with remote memory using the kernel based swapping. FluidMem [25] supports full memory disaggregation for virtual machines through hogplug memory regions and relies userfaultfd to achieve transparent page fault handling. AIFM [26] integrates swapping with application and operates at object granularity instead of page granularity to reduce network amplification. Semeru [27] provides a JVM based runtime to managed applications with disaggregated memory and offloads garbage collection to servers that holding remote memory. Remote Regions [55] applies file abstraction for remote memory and provide both block (`read()`/`write()`) and byte (`mmap()`) access interface. In this paper, we implement a fully user-space swapping framework and co-design it with LWT for data-intensive and high-currency applications.

Distributed share memory (DSM). DSM systems [56–61] provide an unified abstraction by exposing an shared global address space to applications. Different from remote memory, DSM provides an memory abstraction that data is shared across different hosts, therefore bringing significant cache coherence costs and making DSM inefficiency. To avoid the coherence costs, Partitioned Global Address Space (PGAS) [62–65] is proposed but requires application modification. Lightswap that lets applications transparently utilize remote memory through swapping is more efficient.

7 Conclusion

This paper proposes an user-space swapping mechanism that can fully exploit the high performance and low latency of emerging storage devices, as well as the RDMA-enable remote memory. We focus on three main aspects: 1) how to handle page faults in user-space effectively; 2) how to make user-space swapping both high-performance and application-transparent; 3) how to deal with paging errors which are necessary but not considered in previously works.

References

[1] Memcached. https://memcached.org/. 2021.

[2] VoltDB. https://www.voltdb.com/. 2021.

[3] Mohit Saxena and Michael M. Swift. FlashVM: Virtual memory management on flash. In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC'10)*, 2010.

[4] S. Ko, S. Jun, Y. Ryu, O. Kwon, and K. Koh. A new linux swap system for flash memory storage devices. In *Proceedings of the 2008 International Conference on Computational Sciences and Its Applications*, 2008.

[5] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jin-soo Kim, and Joonwon Lee. CFLRU: A replacement algorithm for flash memory. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'06)*, 2006.

[6] Jeffrey C. Mogul, Eduardo Argollo, Mehul Shah, and Paolo Faraboschi. Operating system support for nvm+dram hybrid main memory. In *Proceedings of the 12th Conference on Hot Topics in Operating Systems (HotOS'09)*, 2009.

[7] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-mei Hwu. Flatflash: Exploiting the byte-accessibility of ssds within a unified memory-storage hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*, 2019.

[8] Viacheslav Fedorov, Jinchun Kim, Mian Qin, Paul V. Gratz, and A. L. Narasimha Reddy. Speculative paging for future nvm storage. In *Proceedings of the International Symposium on Memory Systems (MEMSYS'17)*, 2017.

[9] Jian Huang, Anirudh Badam, Moinuddin K Qureshi, and Karsten Schwan. Unified address translation for memory-mapped ssds with flashmap. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA'15)*, pages 580–591, 2015.

[10] Nae Young Song, Yongseok Son, Hyuck Han, and Heon Young Yeom. Efficient memory-mapped i/o on fast storage device. *ACM Transaction on Storage*, 12(4), 2016.

[11] Anirudh Badam and Vivek S. Pai. SSDAlloc: Hybrid ssd/ram memory management made easy. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)*, 2011.

[12] C. Wang, S. S. Vazhkudai, X. Ma, F. Meng, Y. Kim, and C. Engelmann. NVMalloc: Exposing an aggregate ssd store as a memory partition in extreme-scale machines. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS'12)*, 2012.

[13] X. Ouyang, N. S. Islam, R. Rajachandrasekar, J. Jose, M. Luo, H. Wang, and D. K. Panda. SSD-assisted hybrid memory to accelerate memcached over high performance networks. In *Proceedings of the 2012 41st International Conference on Parallel Processing (ICPP'12)*, 2012.

[14] Intel Corporation. Intel Optane Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html. 2021.

[15] KIOXIA Corporation. Kioxia press release. https://business.kioxia.com/en-us/news/2019/memory-20190805-1.html. 2021.

[16] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *In Proceedings of 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, pages 249–264, 2016.

[17] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. Network support for resource disaggregation in next-generation datacenters. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks (HotNets'13)*, 2013.

[18] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. *SIGARCH Comput. Archit. News*, 37(3):267–278, 2009.

[19] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R. Narasayya. Accelerating relational databases by leveraging remote memory and rdma. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD'16)*, pages 355–370, 2016.

[20] P. S. Rao and G. Porter. Is memory disaggregation feasible? a case study with spark sql. In *Proceedings of the 2016 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS'16)*, pages 75–80, 2016.

[21] HP. The Machine: A new kind of computer. https://www.hpl.hp.com/research/systems-research/themachine/. 2021.

[22] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. System-level implications of disaggregated memory. In *In Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA'12)*, pages 1–12, 2012.

[23] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. *Rethinking Software Runtimes for Disaggregated Memory*, pages 79––92. 2021.

[24] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with INFINISWAP. In *In Proceedings of 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*, pages 649–667, 2017.

[25] Blake Caldwell. *FluidMem: Open Source Full Memory Disaggregation*. PhD thesis, University of Colorado at Boulder, 2019.

[26] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. {AIFM}: High-performance, application-integrated far memory. In *Proceedings of the 14th {USENIX} Symposium on Operating Systems Design and Implementation (OSDI'20)*, pages 315–332, 2020.

[27] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A memory-disaggregated managed runtime. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, pages 261–280, 2020.

[28] Intel Corporation. Storage Performance Development Kit. https://spdk.io/. 2021.

[29] Daniel Borkmann. On getting tc classifier fully programmable with cls bpf. *Proceedings of netdev*, 1, 2016.

[30] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, pages 54–66, 2018.

[31] Melvin E. Conway. Design of a separable transition-diagram compiler. *Commun. ACM*, 6(7):396–408, 1963.

[32] Ana Lúcia De Moura and Roberto Ierusalimschy. Revisiting coroutines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(2):1–31, 2009.

[33] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin Levandoski, and Gor Nishanov. Exploiting coroutines to attack the "killer nanoseconds". *Proc. VLDB Endow.*, 11(11):1702–1714, 2018.

[34] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. Interleaving with coroutines: A practical approach for robust index joins. *Proc. VLDB Endow.*, 11(2):230–242, 2017.

[35] NVM Express Work Group. NVM Express. https://nvmexpress.org/. 2021.

[36] DPDK community. DPDK Home. https://www.dpdk.org/. 2021.

[37] W. Cao and L. Liu. Hierarchical orchestration of disaggregated memory. *IEEE Transactions on Computers*, 69(6):844–855, 2020.

[38] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. Software-defined far memory in warehouse-scale computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*, pages 317–330, 2019.

[39] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, and O. Mutlu. Error characterization, mitigation, and recovery in flash-memory-based solid-state drives. *Proceedings of the IEEE*, 105(9), 2017.

[40] Vincent M Weaver. Linux perf_event features and overhead. In *Proceedings of the 2nd International Workshop on Performance Analysis of Workload Optimized Systems (FastPath'13)*, volume 13, page 5, 2013.

[41] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys'20)*, 2020.

[42] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. Memory errors in modern systems: The good, the bad, and the ugly. *ACM SIGPLAN Notices*, 50(4):297–310, 2015.

[43] The kernel development community. Userfaultfd. https://www.kernel.org/doc/html/latest/admin-guide/mm/userfaultfd.html. 2021.

[44] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*, pages 143—-154, 2010.

[45] Tia Newhall, Sean Finney, Kuzman Ganchev, and Michael Spiegel. Nswap: A network swapping module for linux clusters. In *Proceedings of the European Conference on Parallel Processing (Euro-Par'03)*, pages 1160–1169, 2003.

[46] Sandhya Dwarkadas, Nikolaos Hardavellas, Leonidas Kontothanassis, Rishiyur Nikhil, and Robert Stets. Cashmere-vlm: Remote memory paging for software distributed shared memory. In *Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP'99).*, pages 153–159, 1999.

[47] Michael J Feeley, William E Morgan, EP Pighin, Anna R Karlin, Henry M Levy, and Chandramohan A Thekkath. Implementing global memory management in a workstation cluster. In *Proceedings of the fifteenth ACM Symposium on Operating Systems Principles (SOSP'95)*, pages 201–212, 1995.

[48] Evangelos P Markatos and George Dramitinos. Implementation of a reliable remote memory pager. In *Proceedings of the USENIX Annual Technical Conference (ATC'96)*, pages 177–190, 1996.

[49] Shuang Liang, Ranjit Noronha, and Dhabaleswar K Panda. Swapping to remote memory over infiniband: An approach using a high performance network block device. In *Proceedings of the 2005 IEEE International Conference on Cluster Computing (ICCC'05)*, pages 1–10, 2005.

[50] H. Oura, H. Midorikawa, K. Kitagawa, and M. Kai. Design and evaluation of page-swap protocols for a remote memory paging system. In *Proceedings of the 2017 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM'17)*, pages 1–8, 2017.

[51] Hiroko Midorikawa, Yuichiro Suzuki, and Masatoshi Iwaida. User-level remote memory paging for multithreaded applications. In *Proceedings of the 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid'13)*, pages 196–197, 2013.

[52] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, pages 69–87, 2018.

[53] Amanda Carbonari and Ivan Beschasnikh. Tolerating faults in disaggregated datacenters. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks (HotNets'17)*, pages 164–170, 2017.

[54] Luiz Andre Barroso. Warehouse-scale computing: Entering the teenage decade. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA'11)*, 2011.

[55] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote regions: a simple abstraction for remote memory. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC'18)*, 2018.

[56] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles (SOSP'91)*, pages 152–164, 1991.

[57] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, 1989.

[58] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, 1991.

[59] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'96)*, page 174–185, 1996.

[60] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC'15)*, pages 291–305, 2015.

[61] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC'17)*, page 323–337, 2017.

[62] Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[63] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L. Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, Costin Iancu, Amir Kamil, Rajesh Nishtala, Jimmy Su, Michael Welcome, and Tong Wen. Productivity and performance using partitioned global address space languages. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation (PASCO'07)*, page 24–32, 2007.

[64] Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter. Partitioned global address space languages. *ACM Computer Surveys*, 47(4), 2015.

[65] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 519–538, 2005.