Hermit: Low-Latency, High-Throughput, and Transparent Remote Memory via Feedback-Directed Asynchrony

Yifan Qiao[†*] Chenxi Wang[†◇] Zhenyuan Ruan[‡] Adam Belay[‡] Qingda Lu[♯]
Yiying Zhang[§] Miryung Kim[†] Guoqing Harry Xu[†◇]
[†]*UCLA* [‡]*MIT CSAIL* [♯]*Alibaba Group* [§]*UCSD*

Abstract

Remote memory techniques are gaining traction in datacenters because they can significantly improve memory utilization. A popular approach is to use kernel-level, page-based memory swapping to deliver remote memory as it is transparent, enabling existing applications to benefit without modifications. Unfortunately, current implementations suffer from high software overheads, resulting in significantly worse tail latency and throughput relative to local memory.

Hermit is a redesigned swap system that overcomes this limitation through a novel technique called *adaptive, feedback-directed asynchrony*. It takes non-urgent but time-consuming operations (*e.g.*, swap-out, `cgroup` charge, I/O deduplication, *etc.*) off the fault-handling path and executes them asynchronously. Different from prior work such as Fastswap, Hermit collects runtime feedback and uses it to direct how asynchrony should be performed—*i.e.*, whether asynchronous operations should be enabled, the level of asynchrony, and how asynchronous operations should be scheduled. We implemented Hermit in Linux 5.14. An evaluation with a set of latency-critical applications shows that Hermit delivers low-latency remote memory. For example, it reduces the 99$^{\text{th}}$ percentile latency of Memcached by **99.7% from 36 ms to 91 µs**. Running Hermit over batch applications improves their overall throughput by **1.24**× on average. These results are achieved without changing a single line of user code.

1 Introduction

Techniques enabling datacenter applications to use remote memory [10, 17, 28, 29, 36, 43, 52, 53, 57] have gained traction due to their potential to break servers' memory capacity wall, thereby significantly improving datacenters' resource utilization. Compared to clean-slate techniques [17, 52] that provide new primitives for developers to efficiently manage remote memory, swap-based techniques [3, 10, 29, 53, 57, 58] that piggyback on existing paging/swap mechanisms in the OS kernel are more practical as they offer transparency, allowing legacy code to run *as is* on a far-memory system.

The main drawback of swap-based remote access is the overhead incurred by the kernel's paging system. For example, when running Memcached using Fastswap [10], the current state-of-the-art swapping system for Linux, a remote access takes an average of *14 µs*, of which only *9 µs* are spent on network (RDMA) operations—the software-induced overhead is *above 50%*! This large fault-handling overhead significantly increases operation latency, precluding the use of remote memory with latency-critical applications.

In addition, long remote-access time can further block subsequent instructions dependent on these accesses, leading to substantial reductions in application throughput. For example, the performance of garbage collection in a managed language runtime is highly sensitive to memory access latency due to its pointer-chasing nature. Reductions in GC performance can lead to delayed object creations, dramatically reducing the application's overall throughput [42, 57, 58].

The underlying reason for such high overhead is a mismatch in the design of today's swap-based paging systems, which originally targeted slow, disk-based storage, and modern datacenter networks (*e.g.*, 100-400 GbE) that can deliver pages much faster. For example, through profiling, we reveal the following performance bottlenecks that persist in Linux (§3):

- **Page reclamation blocks the critical path**: To make room to fault in new pages, the OS must reclaim memory by swapping out cold pages. Linux is designed to handle this asynchronously by swapping out pages in a separate thread. However, when Linux fails to keep up with the demand for new pages, the page fault handler must block and wait for reclamation to finish.
- **Duplication checks are too conservative**: Linux is designed to never make duplicate I/O requests for the same page. Although this occasionally prevents wasted bandwidth, it comes at a high cost in terms of synchronization overhead, such as during swap cache lookup and insertion.
- **Opportunities for batching are not exploited**: Batching can be an effective optimization when it does not harm page fault handling latency. For example, when Linux performs page reclamation, it first selects a set of victim pages and then swaps out each page individually. A better strategy would be to process victim pages in batches, reducing the cost of TLB shootdowns, I/O writes, and `cgroup` accounting.

State of the Art. The conventional wisdom is that software overheads can be overcome by bypassing the ker-

[*] Part of the work was done when Yifan Qiao interned at Alibaba Group.
[◇] Corresponding authors.

nel [48, 52, 63]. This approach typically requires application-level modifications or the use of custom APIs, making it impractical to deploy transparently across all applications. Our aim is to answer the following question instead: *Can we eliminate performance bottlenecks in the kernel directly, allowing the benefits of fast remote memory to be exposed to all applications transparently?*

Recent work, such as Fastswap [10] and InfiniSwap [29], has made some progress in optimizing the kernel's swap subsystem, such as the use of RDMA to deliver remote pages more efficiently. Fastswap, the current state-of-the-art, also modifies the Linux Kernel to offload page reclamation to a dedicated core and executes it asynchronously. This increases swap-out efficiency, and reduces the time that a page fault handler must block waiting for reclamation to finish. However, Fastswap leaves other opportunities for asynchrony on the table. In addition, a single, dedicated core is insufficient to accommodate changes in demand for swap-out throughput under time-varying memory pressure, limiting the conditions where Fastswap can perform well (§3).

Insights. This work builds on three insights, all centering around asynchrony. First, asynchrony can be used to reduce the latency of page fault handling. For example, during a page fault, the kernel first looks for the page in the swap cache. If the page is present, it will be mapped at the faulting address and the kernel does not need to issue a fetch. However, this check is protected by a lock, which incurs a non-trivial overhead. Instead, fetching a page via RDMA, even if the page is already in the swap cache, is extremely fast: its only penalty is slightly wasted network bandwidth (*i.e.*, bandwidth is rarely saturated). By always issuing the fetch asynchronously and overlapping it with the check, we can reduce the fault-handling latency.

Second, only page faults handlings are latency critical, so it is safe to aggressively optimize all other operations for throughput via batching. For instance, when TLB shootdowns are batched, it reduces the number of interrupts that have to be sent across cores. As another example, RDMA writes of multiple swapped-out dirty pages can be batched into a single transfer. These opportunities are only possible because such operations are conducted asynchronously; otherwise, batching would delay critical swap-in operations.

Third, to achieve optimal performance, the use of asynchrony (e.g., number of cores) must be adjusted dynamically. For example, it is critical that swap-out throughput is perfectly balanced with swap-in throughput. If swap-out throughput is too low, the page fault handler will block and delay the application. If it is too high, it will leave a substantial portion of local memory underutilized, impacting application performance. This is especially challenging because the swapping rate depends on the workload, its inputs, and even the different phases within its execution.

Hermit. This paper presents Hermit, a new paging/swap system that exploits these (previously-unknown) opportunities for asynchrony. Hermit employs *feedback-directed asynchrony* as the major principle in the paging system design, simultaneously enabling full code transparency (*i.e.*, any legacy code can run *as is*), low remote access latency, and high application throughput. Hermit employs different types of asynchrony to tackle the three bottlenecks (*i.e.*, blocked swap-ins, conservative checks, lack of batching), as elaborated below:

First, page reclamation is moved into a set of reclaim threads, which eagerly evict (least-recently used) pages and aggressively batch expensive operations involved in each swap-out (§4.2). In particular, Hermit batches page unmapping, TLB shootdown, RDMA writes, polling, and `cgroup` uncharging in swap-out threads, reducing the amounts of computation involved in swap-outs and improving their throughput (§4.4).

Second, Hermit opportunistically bypasses the swap-in duplication check and issues I/O read requests eagerly, delaying such checks to the synchronous PTE update stage. Since only one thread can successfully update the PTE, all other competing threads will eventually release their duplicate pages, guaranteeing safety (§4.3).

Third, inspired by optimistic locking [4], Hermit makes page I/O fully asynchronous during swap-in to further reduce latency. We split the swap-in procedure into two components: one that can still successfully run and is reversible even if there are concurrent updates, and a second that may either abort or create irreversible side effects in the presence of concurrent updates. Hermit moves the first component out of the critical section to overlap it with the page I/O (details are in Figure 4). Hermit checks the validity before the critical section finishes (*i.e.*, whether concurrent updates have occurred) and if they have, reverts the speculatively executed operations.

Finally, we create a feedback control system for each type of asynchronous operation, using execution profiles to adjust whether and how asynchrony should be applied. In particular, we use (1) *page turnaround* (*i.e.*, time between a page's swap-in and previous swap-out), (2) *page-in/-reclamation throughput*, and (3) *conflict rates* (*i.e.*, how often concurrent updates occur), as metrics to adjust our asynchrony in dealing with reclamation timing, reclamation intensity, eager swap-in, conservative checks, respectively. Hermit profiles and collects these signals throughout the execution to dynamically adapt to the application's changing behaviors.

Results. Hermit was implemented in Linux 5.14 (released August 2021). We have carefully inspected all relevant kernel patches made since then and confirmed that none of them are directly related to Hermit.

We evaluated Hermit with a set of real-world applications including both latency critical (Memcached, SocialNet, and Gdnsd) and batch processing applications (Apache Spark, XGBoost, and Apache Cassandra). Our evaluation on Memcached demonstrates that Hermit outperforms Fastswap [10]

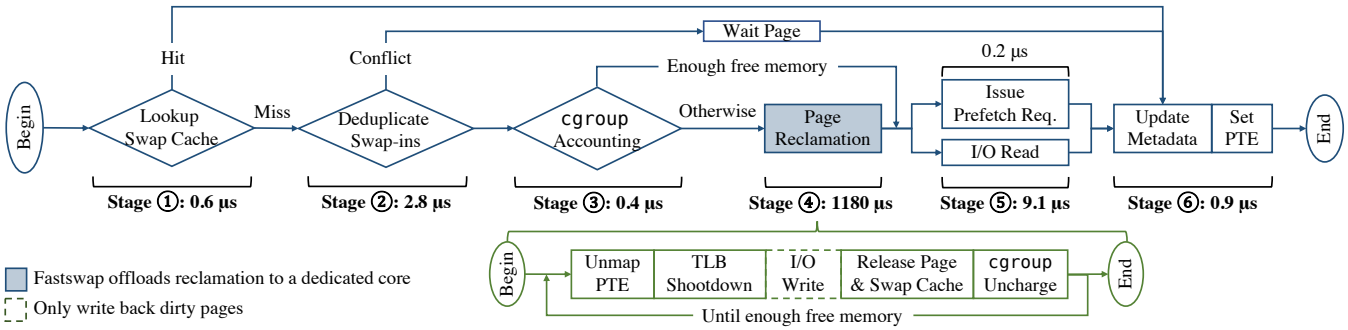


Figure 1: The life cycle of a remote memory page fault in Linux swap.

by **99.7%** in latency, reducing the 99th percentile latency **from 36 ms to 91 µs**. For batch processing applications, Hermit improves throughput by up to **1.87**× with a geometric mean of **1.24**×. Hermit also scales much better with the number of cores than Fastswap. These results demonstrate that low tail latency and high throughput can be achieved at the same time without bypassing kernel, making Hermit a practical solution for enabling remote memory. Hermit is available at https://github.com/uclasystem/hermit.

2 Background

Today's datacenter applications expose high load variability and diurnal patterns [13, 14, 47]. Despite low average load, operators have to provision resources for peak demand to avoid violating service-level agreements. Memory is an especially challenging resource because it is *uncompressable*, meaning that running out of it causes tasks to be killed, which can be very disruptive to overall performance [56]. This is a major contributing factor to low memory utilization in today's datacenters [41, 56].

Remote memory offers a promising solution to improving memory utilization. Its key idea is to break the hardware boundary and unstrand the idle memory of remote machines through fast datacenter networking. Existing systems have demonstrated the feasibility of utilizing remote memory with good performance [10, 17, 52]. Among different approaches to realizing remote memory, the kernel-based approach offers a unique advantage of transparency. It enables existing applications to run as is over remote memory using commodity hardware. This is very attractive to datacenter operators as it significantly lowers the bar for adoption.

The kernel-based approach achieves transparency through paging, an idea that dates back to the 1960s. Originally, paging was designed to extend the addressable memory space with a slow but large secondary storage (usually a mechanical disk). Under memory pressure, the kernel pages out cold pages to disk and marks them as absent from memory. Later, if a process accesses any of those pages, the memory management unit (MMU) raises a page fault exception which transparently traps the control flow into the kernel to page in the data and update the corresponding page table entry (PTE).

Linux implements paging in its swap subsystem, which is often used as the last resort for preventing out-of-memory (OOM) killing. Swap can serve as a temporary mechanism that buys operators time to solve memory pressure, *e.g.*, by migrating or killing processes. The architecture of the paging/swap subsystem has remained relatively stable since its inception. However, in the context of remote memory, fast network-attached memory (4 µs, 12 GB/s) can be used as a secondary storage device as opposed to a slower disk (10 ms, 200 MB/s). Due to this huge performance gap, the legacy swap system is a bottleneck in accessing remote memory. For example, when running Memcached on Fastswap (*i.e.*, the state-of-the-art swap system) with a high local memory ratio (70%), we see a 4× throughput drop.

3 Understanding Existing Swap Systems

3.1 The Life Cycle of Remote Memory Access

The legacy design of Linux swap imposes high overheads on accessing remote memory. To better understand the root cause of its inefficiencies, we conducted a performance study by running Memcached on Fastswap [10] (the state-of-the-art swap system). Figure 1 shows the stages of a remote memory access and breaks down their costs. We discuss each stage in more detail as follows:

① Lookup swap cache. The swap cache serves as a centralized component that prevents race conditions. It tracks the information of swapped-in pages and ongoing swap-out requests. First, the faulting page may have been fetched by another process or the OS prefetcher. By looking up the swap cache, Linux detects this and jumps to stage ⑥. Second, it is possible that the faulting page is being swapped out by another process. In this case, naïvely fetching the remote page will see the stale copy. With the swap cache, Linux detects the race and cancels the ongoing swap-out. Looking up the swap cache takes an average of 0.6 µs.

② Deduplicate swap-ins. At the same time, there can be multiple threads swapping in the same page. Linux guarantees that only one thread can succeed by synchronizing with lock primitives. The remaining threads will be busy waiting until the page gets fetched. This design saves I/O bandwidth

but impacts latency and hurts scalability. This stage takes an average of 2.8 µs.

③ `cgroup` accounting. Before fetching the page, Linux must ensure that the current process has sufficient free memory by performing `cgroup` accounting. For the lucky process with enough memory, it jumps to stage ⑤ directly. The accounting stage takes an average of 0.4 µs. Otherwise, Linux must go through stage ④ to reclaim pages to make room, as elaborated below.

④ Direct page reclamation. Linux iteratively reclaims pages until the size of the available local memory is above the low-water mark. Linux swaps out a single page for each iteration. Swap-out is expensive as it involves operations such as TLB shootdown, PTE unmapping, *etc.* This stage exists only when the local memory runs low, but it is also the longest one that takes an average of 1180 µs. To reduce direct reclamation, Fastswap performs this stage asynchronously with a dedicated core.

⑤ Fetch and prefetch page. Linux issues an I/O request to fetch the faulted page. Meanwhile, it may issue multiple prefetching requests. This stage takes an average of 9.1 µs.

⑥ Update metadata. Finally, Linux updates kernel metadata, including page table entries (PTEs), swap entries, and page reverse mapping (`rmap`). This stage takes 0.9 µs.

3.2 Root Causes of Inefficiencies

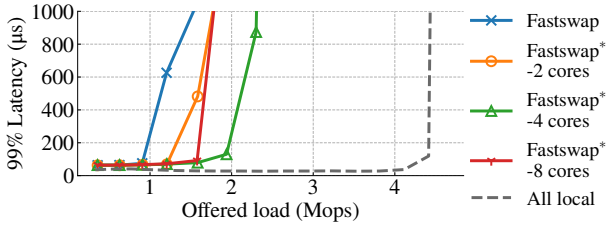


Figure 2: 99th percentile latency with respect to offered load of Memcached on Fastswap under 70% local memory.

To understand the bottleneck imposed by Fastswap's single, dedicated reclamation core, we ran several experiments with Memcached. Figure 2 shows Memcached's 99th percentile latency with respect to its offered load when running with 70% local memory. The baseline for comparison is Memcached running locally (100% local memory without swapping), which is the rightmost curve and achieves >4.4 Mops load throughput with good tail latency. Memcached on Fastswap (the blue curve), however, can only offer ≈1 Mops load before the dedicated core gets saturated and its latency increases dramatically. The reason is that Fastswap's single dedicated core cannot keep up with the increasing demand for page reclamation. We then modified Fastswap's original implementation to offload page reclamation onto multiple cores, denoted as `Fastswap*` in the figure, as a naïve strawman approach.

Using more dedicated cores can indeed help reduce the direct reclamation ratio, as shown in Figure 3. With 4 dedicated cores, `Fastswap*` is able to eliminate direct page

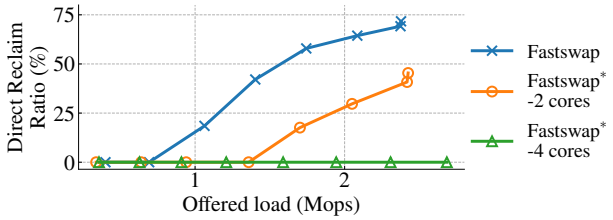


Figure 3: Direct page reclamation ratio of Memcached on Fastswap under 70% local memory.

reclamation, thus providing the highest throughput among all Fastswap variants. However, Fastswap uses static core provisioning, which is insufficient in practice due to the phased behaviors and shifts in load that occur within datacenter applications. First, the number of required dedicated cores depends on the application's working set, the available local memory, and the swap-in intensity, making it impossible for a statically determined number to work universally for different applications or even different phases of the same application. Second, over-provisioning dedicated cores does not always lead to greater end-to-end performance; in many cases, using more cores only shifts the bottleneck from page reclamation to the application itself, as more dedicated cores for reclamation imply fewer available cores for application threads. As shown in Figure 2, increasing the number of dedicated cores in Fastswap from 1 to 4 (`Fastswap*-4 cores`) improves performance, but further allocating cores degrades performance (`Fastswap*-8 cores`). Furthermore, although `Fastswap*-4 cores` eliminates direct page reclamation (*i.e.*, reducing latency), it still loses ∼45% performance (*i.e.*, reducing throughput). The performance loss is due to three major kinds of inefficiencies induced by Linux swap, as elaborated below.

Swap-out blocks swap-in. As explained earlier, Memcached experiences high memory access latency when running short of local memory, as it has to reclaim pages. Page reclamation is expensive as it requires finding victim pages and unmapping them, followed by a number of expensive operations for consistency such as TLB shootdown. This significantly impacts its tail latency, leading to violations of the service-level agreement (SLA).

Fastswap tackles this issue by allocating a dedicated core to reclaim pages asynchronously in the background. However, as discussed earlier, it is nearly impossible to statically identify the optimal number of cores due to load variability.

Unoptimized for fast I/O. Linux swap was designed for slow secondary storage like hard-disk drives whose performance is two to three orders of magnitude lower than today's remote memory in both bandwidth and latency. Since disk bandwidth is often the bottleneck, Linux applies aggressive optimizations in its page fault handling path to reduce I/O traffic (stage ②). While they were effective in the era of slow disks, these optimizations become irrelevant in the context of remote memory whose bandwidth is close to the bandwidth of main memory. Even worse, the outdated op-

timization generates an adversarial performance impact; it prolongs remote memory access latency, hurting scalability (*e.g.*, due to synchronization). For latency-critical applications like Memcached, prolonged remote memory accesses can significantly increase the time for serving incoming requests, imposing super-linear effects on tail latency. Modeled by queueing theory [24], for instance, 10% longer service time can potentially double the 99th percentile latency, leading to vast SLA violations.

Additionally, since the disk latency (ms-scale) is significantly higher than the CPU time in page fault handling (µs-scale), Linux adopts a serial-execution model for simplicity. As shown in Figure 1, the I/O read stage is executed separately from other stages; after issuing the I/O read request, Linux either busy waits for the I/O response or re-schedules the faulting thread (which hurts latency of fast I/O requests), relinquishing the opportunity of overlapping the waiting period with other stages.

Unoptimized for CPU overhead. Linux swap is a mechanism aimed at avoiding OOM killing. Inherently, treating swapping as a rare event, it was designed to optimize for responsiveness, *not* for CPU efficiency. For example, during page reclamation (stage ④), Linux swaps out only one page at a time, under the assumption that by releasing the space more timely it can unblock the OOM process sooner. Unfortunately, this amplifies the CPU usage as it must invoke expensive operations such as TLB shootdown for *every reclaimed page*. While overhead is acceptable when swapping is rare, it grows significantly in the scenario of remote memory (which is swapping-intensive). In the case of Memcached, 12.6% of the total CPU time is spent on reclaiming pages, not on application tasks. To make matters worse, Linux swap heavily relies on locks to synchronize page reclamation and scales poorly. Hence, the overhead will further increase with the number of concurrent swapping operations.

Key takeaway. Linux swap imposes high overheads to remote memory access primarily due to the above three issues. Fastswap, the state-of-the-art swap system, partially tackles the first issue, but neglects the last two. For the first issue, Fastswap uses statically provisioned cores to run swap-out tasks; as shown in Figure 2, static core provisioning cannot adapt to dynamic load changes, leading to either insufficient or wasted CPU resources.

4 Hermit Design

4.1 Design Overview

To overcome the aforementioned inefficiencies, we developed Hermit, a new swap system based on the principle of *feedback-directed asynchrony*. Our key insight is that asynchrony should be used aggressively (to overlap nonurgent and urgent operations to reduce latency), but this must be done in a controlled manner—whenever asynchrony cannot bring benefits, we should switch back to the conventional synchronous design. Figure 4 illustrates Hermit's design.

First, Hermit optimizes tail latency of accessing remote memory by moving page reclamation from the critical path into the background (§4.2). Instead of following the design of Fastswap, which statically reserves a certain number of dedicated cores, Hermit relies on a *reclaim scheduler* to dynamically schedule reclaim threads. The scheduler leverages feedback from `cgroup` counters to determine the right timing and the appropriate number of cores for reclamation.

Second, the swap-in path of Hermit was designed with fast remote memory in mind (§4.3)—for remote memory, it is reasonable to trade off network usage for end-to-end performance as modern datacenter network offers abundant bandwidth (100-400 Gbps). In the common case, Hermit detects idle network bandwidth and opportunistically bypasses swap-in duplication checks (stage ② in §3) to improve scalability and reduce latency. This bypassing has a consequence: in the (rare) case that multiple threads are fetching the same page at the same time, they will all transfer the same page over the network. Note that this will *not* lead to correctness issues because only one copy will be mapped by the PTE in the last stage, and any other requests will abort and release their page. However, it may potentially waste some network bandwidth when duplicate pages are requested. Therefore, instead of bypassing blindly, we use the conflict rate (in the last stage) as a control signal to determine whether it is beneficial to enable bypassing. To further optimize the critical-path latency, Hermit also overlaps the I/O read stage with other swap-in operations (*e.g.*, `cgroup` accounting, metadata updating, *etc.*).

Finally, we structured Hermit to operate in a swap-intensive environment to match the reality of using remote memory (§4.4). Hermit carefully optimizes the CPU usage of page reclamation so that more CPU resources are available for applications. Enabled by Hermit's reclaim scheduler, which reduces the "urgency" of reclamation tasks, Hermit opportunistically handles reclamation requests *in batches* to amortize the overhead. In addition, Hermit bypasses the expensive reverse mapping operation when swapping out a private page (which is common). As a result, Hermit not only reduces the remote access latency but also significantly improves the application's throughput.

4.2 Reclaim Scheduling

In Linux swap, the direct page reclamation in the swap-in path significantly impacts the tail latency of accessing the remote memory. To reduce tail latency, Hermit moves reclamation off the critical path into background threads; the reclaim scheduler monitors the free memory size and *proactively* starts reclamation before memory exhaustion. The scheduler uses the application's swap throughput as a feedback signal to auto-tune the number of reclaim threads.

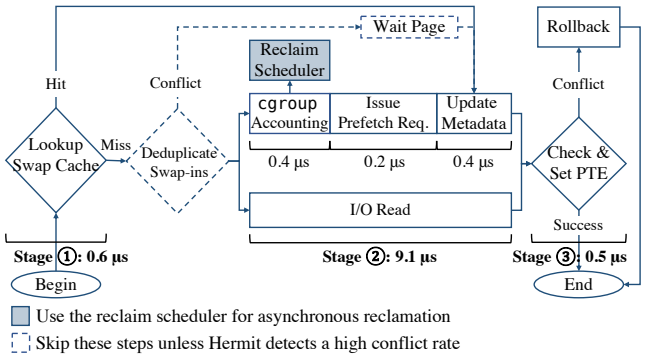


Figure 4: The life cycle of a remote memory page fault in Hermit.

Designing such a scheduler is challenging because it must determine both the right *timing* and the appropriate *amount of CPU resources* for reclamation. (1) As for the timing, if the scheduler starts reclamation too early, a substantial portion of local memory would be underutilized, impacting application performance; on the flip side, if the scheduler starts reclamation too late, the application would exhaust the local memory and suffer from direct reclamation. (2) As for CPU resources, under-provisioning cores for reclamation (*i.e.*, the case of Fastswap) make it unable to keep up with the local memory consumption rate, leading to memory exhaustion, while over-provisioning cores is also undesired as it contends with the application and reduces its performance.

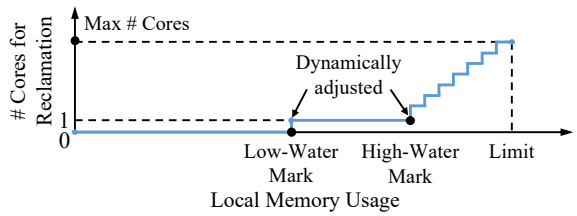


Figure 5: Adaptive reclaim scheduler.

Figure 4 shows the design of the reclaim scheduler, which leverages counters from `cgroup` to schedule reclamation. Since the timing for reclamation is critical to performance, our reclaim scheduler has to be very reactive to free memory size changes (in µs-level). Instead of using a dedicated core to poll the memory usage which waste CPU cycles, Hermit adopts a *decentralized* reclaim scheduler; it inlines the scheduler code into the `cgroup` charging, an indispensable step for swap-ins. This design enables us to discover any sudden change in the free memory size with only a few CPU cycles.

Hermit's scheduling policy follows the conventional wisdom of random early detection [26] to gradually increase its asynchronous reclaim throughput. Specifically, Hermit starts asynchronous reclamation when application's memory budget is running low, but Hermit will only enable a small number of reclaim threads first and gradually increase the number of reclaim threads after observing constantly increasing memory usage. The intention of the design is to handle a burst of swap-ins within the memory limit with as few reclaim threads as possible, and thus minimizing asynchronous reclamation's interference to the application.

On the other hand, when the application is about to run out of memory, Hermit must unleash the full power of asynchronous reclaim threads to match the reclaim throughput to swap-in throughput to avoid direct reclamation, offering the application maximum swap performance. Figure 5 depicts Hermit's adaptive scheduling policy, which determines the number of cores for page reclamation given the application's current local memory usage. The curve can be divided into three phases, marked by the low-water mark and the high-water mark to differentiate the urgency of asynchronous reclamation.

When the application does not swap intensively and its local memory usage is below the low-water mark, the number of reclamation cores is zero, indicating that the asynchronous page reclamation is disabled now to let application threads have all CPU cores. When the application's local memory usage is between the low-water mark and the high-water mark, it indicates that the application is under memory pressure, and the scheduler will assign one core for asynchronous reclamation to relieve the memory pressure with minimal compute to minimize its interference to application's threads.

Finally, when the application hits the high-water mark, it indicates that the application is about to run out of memory. Page reclamation is an urgent task now to prevent the application from triggering direct page reclamation. As such, the reclaim scheduler must assign more cores for reclamation to match the reclaim throughput with application's swap-in throughput. As Figure 5 shows, during this phase, the number of cores assigned for reclamation is proportional to the local memory usage, reaching the maximum value when the local memory usage equals the memory limit.

Hermit leverages the kernel's runtime statistics to autotune the low and high memory watermarks, as elaborated below.

High memory watermark. Hermit dynamically adjusts the high memory watermark based on the application's current *swap intensity*. We define swap intensity as the overall swap-in throughput divided by the per-core page reclamation throughput, representing the number of cores needed for reclamation to match the swap-in speed. Intuitively, when the swap intensity increases, we should lower the high-water mark to start ramping up reclamation earlier; and when the swap intensity decreases, we should raise the high-water mark accordingly. Hermit sets the high-water mark as $MEM\_LIMIT - \alpha \cdot SWAP\_INTENSITY$, where $\alpha = 128$ works well in practice.

Low memory watermark. Initially, Hermit sets the low-water mark to be the same as the high-water mark. Then it gradually probes its optimal value based on the average page turnaround time (APT), defined as the average duration for swapped-out pages to remain untouched. When APT does not increase, Hermit attempts to lower the low-

water mark, as now it can potentially start reclamation earlier without impacting the application performance. However, when APT increases, Hermit immediately raises back the low-water mark to revert the negative impact on the application performance.

4.3 Adapt Swap-in to Fast Remote Memory

As shown in Figure 4, Hermit re-architects the swap-in path for the fast remote memory with two main innovations.

Eager swap-in. Hermit opportunistically bypasses the swap-in duplication check to minimize latency. As such, it is now possible that multiple threads issue swap-in requests for the same page. To ensure that only one of them will succeed, Hermit synchronizes them in the final stage (updating PTE) using a fine-grained lock. All other failed threads will release their swapped-in pages—CPU cycles consumed by them are wasted and considered as penalty. Hermit collects the conflict rate and the penalty as feedback to reassess whether it is still beneficial to enable eager swap-in and disable it if it impacts performance.

Asynchronous I/O. Hermit further shortens the critical path of swap-ins by overlapping the I/O read with other operations, for example, `cgroup` charging. If later the `cgroup` check shows no memory, Hermit discards the I/O read response and updates the failure counter. Hermit falls back into synchronous I/O when the failure ratio is high. This happens very rarely in practice thanks to Hermit's asynchronous reclamation (§4.2).

4.4 CPU-Efficient Page Reclamation

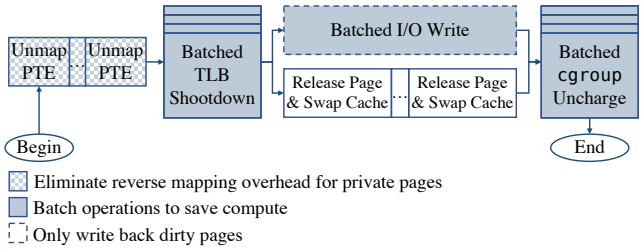


Figure 6: Hermit's asynchronous page reclamation path.

As shown in Figure 6, Hermit carefully optimizes the CPU overheads of page reclamation to minimize its performance impact to applications.

Batched reclamation. As illustrated in §3, Linux's page reclamation is mainly designed for slow disk devices where swapping occurs infrequently—it trades off CPU efficiency for responsiveness by only swapping out one page at a time. However, Hermit overcomes the responsiveness loss with its asynchronous reclamation design, which relaxes the responsiveness requirement of page reclamation, thereby creating opportunities for batching. As depicted in Figure 6, Hermit batches expensive operations, including TLB shootdowns, I/O writes, and `cgroup` accountings—to amortize their overheads in the asynchronous page reclamation path.

Reverse mapping elimination. To avoid race conditions during reclamation, Linux has to ensure that the page is immutable before writing it back to remote memory. Linux achieves this goal by using `rmap` (reverse page mapping) to identify and unmap all the virtual pages mapped to the reclaimed physical page. `rmap` walk is expensive as it involves several memory accesses and lock synchronizations. A key observation in Hermit is that most reclaimed pages are private pages (*i.e.*, only referenced by one virtual page). For private pages, Hermit eliminates the expensive `rmap` walk by inlining the virtual page address into the physical page metadata in Linux. This approach trades a tiny portion of local memory (0.2% in the worst case) for better performance.

5 Implementation

We implemented Hermit atop Linux 5.14, the latest release when we started the project. We have been carefully examining every new release to ensure that no patch is directly related to our techniques. We added or modified 9704 lines of kernel code, mainly re-implementing Linux's swap-in and swap-out code paths.

We built our RDMA-based swap backend atop Fastswap's implementation. The original Fastswap uses Linux's `frontswap` interface which only supports blocking I/O. We extended it with an asynchronous I/O interface to enable asynchronous batched I/O writes during page reclamation.

For the swap-in path, we stored the feedback signals `swap_stats`, used by Hermit to decide whether to bypass the swap-in deduplication, in Linux's process context `mm_struct`. `swap_stats` contains two atomic counters representing the numbers of successful and aborted swap-ins respectively. The page fault handler reads and updates `swap_stats` when swapping in the page.

For the swap-out path, we implemented per-`cgroup` reclaim threads as Linux kernel threads. We stored the feedback signals `swap_ctrl`, used by Hermit to decide the swap-out timing, in Linux's memory cgroup `mem_cgroup`. `swap_ctrl` contains two counters representing the total number of charged pages and reclaimed pages. Hermit updates `swap_ctrl` during `cgroup` charging and page reclamation. The reclaim scheduler reads `swap_ctrl` periodically (per 128 charges in our implementation) to calculate the swap intensity for updating the high-water mark. We use Linux's existing mechanism of tracking the page re-fault distance to calculate the average page turnaround (APT) for updating the low-water mark. Hermit batches 32 pages per NUMA node for its asynchronous page reclamation to keep low amortized overheads while ensuring most reclamations can finish timely (within 1 ms). To batch reclamation while ensuring consistency, we carefully ordered the operations (see Figure 6). Hermit first selects and unmaps a batch of pages, and then issues a single TLB flush before writing all dirty pages to remote memory. After which, Hermit rechecks each page to ensure it remains untouched and free it. Other-

wise, the page must have been faulted on and re-mapped into the process' page table, so Hermit skips freeing this page and returns it back to the application. To bypass the `rmap` walk, we stored the virtual address of private pages using a global array. We did not directly embed the virtual address into Linux's per-page metadata to avoid breaking its cache alignment.

6 Evaluation

Our evaluation seeks to answer the following questions:

1. Can Hermit maintain low tail latency (§6.2) and high throughput (§6.3) while delivering remote memory?
2. How does Hermit's performance compare to standard Linux and Fastswap [10]? (§6.2-§6.3)
3. What contributes to Hermit's better performance? (§6.4)

Setup. We ran experiments in a cluster with one CPU server and one memory server, connected by a 100 GbE network. Each server equips a 24-core AMD 7402P CPU and 128 GB memory. Both Hermit and Fastswap ran on Ubuntu 20.04 with Linux 5.14. For latency-critical applications, we generated load from another server, which connects to the CPU server via a 25 GbE network. We followed common practices to tune these servers for low latency [47], including disabling CPU frequency scaling, machine-check exceptions, and transparent hugepages. We also disabled OS security mitigations as recent CPUs have fixed these vulnerabilities. We enabled hyperthreading as it improves the performance of remote memory systems.

Methodology. We compared Hermit with the ideal system that only uses local memory and the state-of-the-art kernel-based remote memory system, Fastswap [10]. To enable a fair comparison, we also ported Fastswap to Linux 5.14, the same kernel version that Hermit uses.

6.1 Real-world Applications

We used six real-world datacenter applications for evaluation, as shown in Table 1.

Category	Application	Dataset	Size
Latency-Critical	Memcached [7]	Facebook's `USR` [14] like	32M KVs
	SocialNet [27]	Socfb-Penn94 [51]	41.5K nodes, 1.4M edges
	Gdnsd [1]	Custom	75M sites
Batch	Spark [62]	Wikipedia EN [8]	188M points
	XGBoost [21]	HIGGS [15]	21M instances
	Cassandra [9]	YCSB [22]	20M records

Table 1: Applications used in the evaluation.

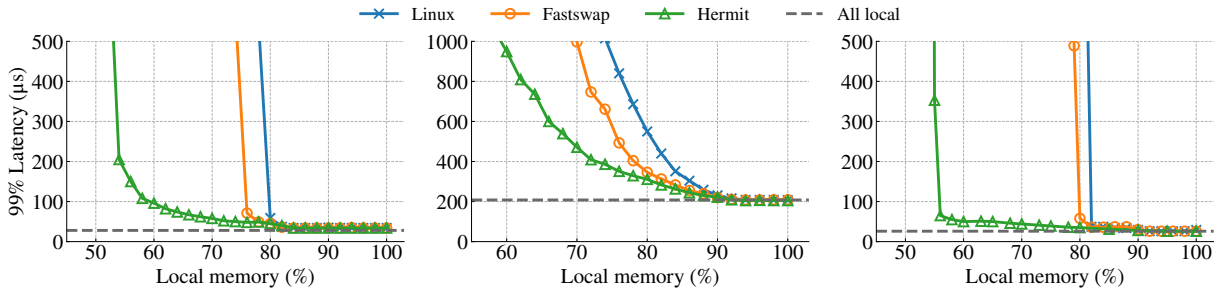
Latency-critical applications. Memcached [7] is a popular in-memory key-value store. It only performs a hash table lookup for each request, leading to a small per-request memory footprint. It has low compute intensity and poor spatial locality. We followed Facebook's `USR` distribution to generate load with 99.8% GET and 0.2% PUT [14]. SocialNet (a part of the DeathStarBench [27]) is a twitter-like interactive web application built with microservices. It has a fan-out pattern in which each client request is served by multiple microservice instances. This leads to a larger per-request memory footprint than Memcached. It has medium compute intensity and poor spatial locality. We rewrote DeathStarBench's python-based load generator using C++ to increase its throughput. Gdnsd is an authoritative-only DNS server. It performs a tree lookup for each DNS query. It has a small per-request memory footprint and low compute intensity. Different from previous applications, Gdnsd has good spatial locality. We generated queries with random domain names for evaluation. For all three applications, we generated requests with keys followed Zipf distribution using the skewness parameter $s = 0.99$, to be consistent with the standard YCSB benchmark suite [22].

Batch applications. Apache Spark [62] is a big data analytics engine. We used the logistic regression model from its official example suite for evaluation, in which Spark trains the model iteratively by scanning the dataset to update the model parameters. It has high compute intensity and a large memory footprint. XGBoost is a gradient boosting library for machine learning. We ran binary classification for evaluation. It initializes a group of decision trees and trains them iteratively by splitting the tree leaves with input data. It has dynamic parallelism and a medium memory footprint. Apache Cassandra [9] is a large-scale NoSQL database. It uses a storage structure similar to a log-structured merge tree, which has medium compute intensity and good spatial locality. Different from other batch applications, it also periodically persists in-memory data to disk. We used YCSB [22] as its workload for evaluation. Both Spark and Cassandra are Java-based and run atop OpenJDK-11. Java's garbage collection makes them more memory intensive. XGBoost is a native C++ application.

6.2 Tail Latency of Latency-Critical Applications

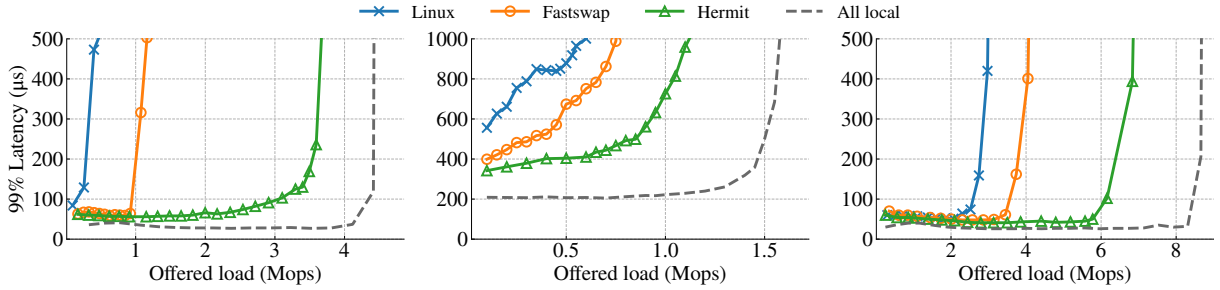
To better quantify the tail latency overhead introduced by Hermit, we use low-latency applications enabled by Shenango (a recent datacenter library OS) [47], for evaluation. With Shenango's low-latency threading runtime and network stack, these applications achieve sub-millisecond tail latency, making it an extremely challenging case for swap systems. We also rerun the same applications with their vanilla (Linux-based) versions. The results (in Appendix A) show similar trends but with higher tail latency for all systems, including the ideal local-only case. This stems from the higher overhead of the kernel's threading and network stack. Following previous studies [34, 49, 63], we primarily focus on applications' 99$^{\text{th}}$ percentile latency in our evaluation. The results of other percentiles (including median and 99.9$^{\text{th}}$) can be found in Appendix C.

We first ran applications with a fixed load (50% of load capacity measured with only using local memory) and varying



(a) Memcached (2 Mops) (b) SocialNet (0.75 Mops) (c) Gdnsd (4 Mops)

Figure 7: Hermit significantly outperforms Fastswap and Linux in terms of 99% latency under the same fixed load and varying local memory ratio. Hermit enables applications to operate in a more challenging regime of less local memory while still maintaining < 500 µs 99% latency.



(a) Memcached (b) SocialNet (c) Gdnsd

Figure 8: Hermit achieves significantly lower 99% latency than Fastswap and Linux under the same fixed local memory ratio and varying load. For Memcached and Gdnsd, Hermit achieves 99% latency close to the ideal local-only case. SocialNet is more challenging due to its higher per-request memory footprint, but Hermit still achieves 74% load capacity of the ideal case.

local memory ratios. We measured the application performance on Linux, Fastswap, Hermit, and the ideal setup that only uses local memory (see Figure 7). The original Linux does not have an RDMA-based swap backend. To enable a fair comparison, we extended it to use Fastswap's RDMA backend. On Figure 7, the X-axis shows the ratio of the local memory provisioned; the Y-axis shows the 99th percentile latency achieved by Linux, Fastswap, Hermit, and the ideal setup.

Intuitively, both Fastswap and Hermit achieve ideal performance when only using local memory. When we decrease the local memory ratio, latency increases as remote accesses become more frequent. However, Hermit's latency increases slower than Fastswap, revealing it is more tolerant to remote accesses. This is because Hermit's overhead of accessing remote memory is lower, thanks to its shorter swap-in path and its reclaim scheduler that eliminates direct reclamation (§6.4.1). As Hermit adaptively changes the number of reclaim threads to match the reclamation rate with the swap-in rate, it can result in competition for CPU resources if the local memory ratio is small enough. Eventually, both systems encounter a "hockey-stick" when they cannot handle the excessive remote memory accesses. Compared to Fastswap, Hermit enables applications to operate in a more challenging regime of less local memory while still maintaining < 500 µs 99th percentile latency.

Specifically, the low compute intensity of Memcached and Gdnsd aligns with Hermit's optimizations well; they only require a few CPU cores for serving load, leaving the rest of the cores for reclamation. Moreover, thanks to their small per-request memory footprints, they only require a small number of reclaim threads. For Memcached, Hermit has to rely on more than four reclaim threads to keep up with frequent swap-ins when Memcached runs under < 60% local memory ratio. The CPU contention gets more severe when local memory gets smaller, and the system reaches 70% CPU utilization under 58% local memory ratio. Afterward, Hermit's reclaim threads can heavily interfere and block Memcached's threads, thus ramping up the tail latency. Similarly, Gdnsd on Hermit used ∼72% CPU cycles when running under 56% local memory ratio, and the system can no longer maintain low 99th percentile latency afterward. Fastswap's single dedicated core fails to keep up with the increasing page reclamation demand when local memory ratio is lower than 76% and 82% for Memcached and Gdnsd, respectively, which ramps up their 99th percentile latency. To conclude, Hermit pushes the operating regime in terms of local memory ratio from 75% (*i.e.*, Fastswap) to 55% for Memcached, and from 80% to 55% for Gdnsd. Gdnsd has a slightly better result due to its better spatial locality. SocialNet is a more challenging application that has a higher compute intensity and a larger per-request memory footprint. It requires more reclaim threads which compete with application threads more heavily under low local memory ratios. The system used 70% of its CPU resources under 65% local memory ratio, and saturated all CPU cores under 60% local memory ratio. Hermit pushes its

regime from 75% local memory ratio to 65%. In summary, Hermit enables applications to store an average of 20% more working set in remote memory without breaking the tail latency target, thereby harnessing stranded memory resources more efficiently.

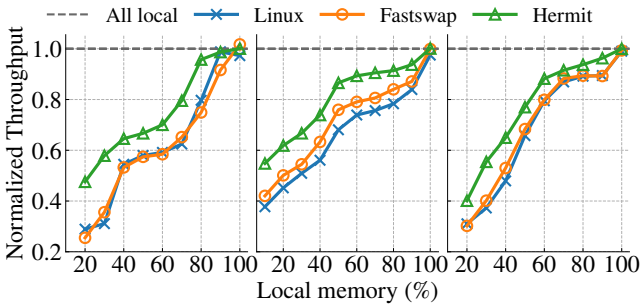
Next, we fixed the local memory ratio to 70% and measured the tail latency of applications with varying load (see Figure 8). Under low load, both Fastswap and Hermit encounter higher latency than the local-only case due to additional remote memory accesses. Hermit delivers lower latency than Fastswap due to the cheaper remote accesses it offers. For Memcached and Gdnsd whose per-request memory footprint is smaller, Hermit reduces 99th percentile latency by 3–9 µs, whereas for SocialNet, Hermit reduces latency by 43–86 µs.

Under high load, the latency gap becomes wider because of the CPU contention between application and asynchronous reclaim threads. In this case, application threads access remote memory intensively, therefore triggering memory reclamation frequently. The asynchronous reclaim threads impact application performance by contending CPU resources. Hermit experiences lower performance degradation because of its asynchronous and more CPU-efficient design of memory reclamation (§6.4.2). By eliminating blocking induced by direct reclamation and shifting more CPU resources from reclamation to application, Hermit handles higher load than Fastswap under the same local memory ratio while still maintaining $<$ 500 µs 99th percentile latency. Hermit improves the load capacity by 3.2× (from 1.1 Mops to 3.5 Mops) for Memcached, and 1.7× (from 4.0 Mops to 6.8 Mops) for Gdnsd. Notably, compared to the ideal local-only case, Hermit enables these applications to enjoy the benefit of remote memory with only an average of 20% decrease in their load capacity. It is more challenging to handle SocialNet well due to its larger per-request memory footprint and higher compute intensity. As a result, the number of reclaim threads needed increases quickly with the load, deteriorating the contention with application threads. Even though, Hermit still improves SocialNet's capacity by 1.5× (from 0.75 Mops to 1.15 Mops).

6.3 Throughput of Batch Applications

In this section, we evaluate the throughput of batch applications under varying local memory ratios (see Figure 9). Hermit outperforms both Fastswap and Linux. It only requires 45%–70% local memory to achieve at least 80% of the ideal throughput for all applications. In contrast, Fastswap (*i.e.* the better baseline) has to use an average of 20% more local memory to achieve the same throughput. Even under the extremely challenging case of 20% local memory, Hermit is still able to preserve 40%–60% of applications' ideal throughput. This leads to 1.23×–1.87× improvement over Fastswap.

When Spark runs atop Fastswap, its throughput drops significantly when running with $<$ 40% local memory. Our profiling reveals that swapping becomes extremely frequent in this case, triggering the scalability bottleneck in kernel's page reclamation path. Hermit does not suffer from the same issue due to two reasons. First, Hermit significantly reduces the direct reclamation ratio by performing reclamation asynchronously and timely. Therefore, it confines reclamation into a small number of reclaim threads rather than all the application threads (in direct reclamation). Second, Hermit's CPU-efficient reclamation design reduces the number of threads needed, further alleviating the scalability issue.



(a) Spark (68.4s) (b) XGBoost (42.2s) (c) Cassandra (72.6s)

Figure 9: We measured the throughput of batch applications achieved by different swap systems normalized to the ideal local-only setup. Hermit outperforms other baselines. The number in the parenthesis shows the ideal execution time.

6.4 Design Drill-Down

We now evaluate specific aspects of Hermit's design to understand their individual contributions to overall performance.

6.4.1 Remote Memory Access Latency

Hermit reduces remote memory access latency by shortening the critical path of swap-ins. Figure 10 breaks down the improvements brought by specific optimizations, including bypassing deduplication and using asynchronous I/O. The results are measured using Memcached. Without Hermit's optimizations, the original Linux spends 2.8 µs on swap-in deduplication. Hermit eliminates this overhead entirely by opportunistically bypassing the deduplication, see Figure 11. After enabling asynchronous I/O, Hermit further overlaps I/O read with other swap-in operations (*e.g.*, `cgroup` accounting and metadata updating), reducing the swap-in latency by another 0.9 µs. With both optimizations turned on, Hermit reduces the page fault handling latency by 35%, from 13.8 µs to 10.2 µs. The RDMA backend spends 9 µs on performing a 4KiB-page I/O. This indicates that Hermit reduces the overhead of the swap system by a factor of four, from 4.8 µs to only 1.2 µs.

6.4.2 Page Reclamation Efficiency

To demonstrate Hermit's improvements on page reclamation efficiency, we ran Memcached and measured the per-thread

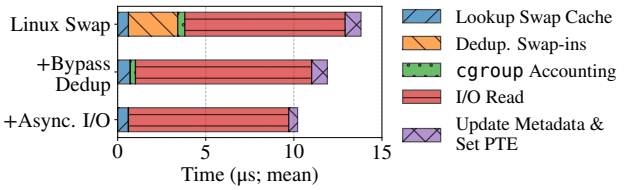


Figure 10: Hermit reduces the remote memory access latency in Memcached from 13.8 µs to 10.2 µs with two optimizations, *i.e.*, bypassing deduplication and using asynchronous I/O.

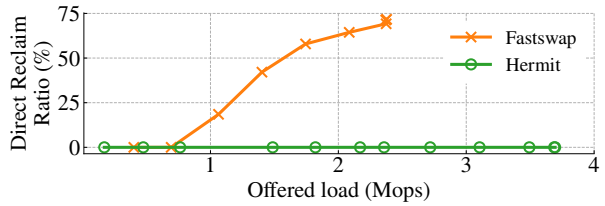


Figure 11: Hermit entirely eliminates direct reclamation for Memcached, thanks to its asynchronous reclamation design. Fastswap fails to serve > 2.4 Mops load due to CPU congestion.

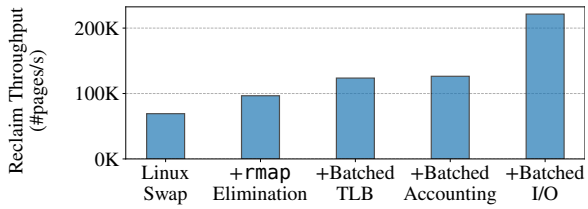


Figure 12: Eliminating reverse mappings and enabling more batching makes reclamation 2.9× more efficient.

reclamation throughput, see Figure 12. As shown by the leftmost bar, the original Linux achieves 77K pages/s reclamation throughput. Hermit's `rmap` elimination optimization effectively improves the throughput by 37%, as most of pages are private in Memcached. Batching TLB shootdowns and `cgroup` accountings amoritizes their overheads and brings an additional 27% and 3% improvement, respectively. Finally, Hermit batches I/O writes for dirty pages and overlaps them with the page release phase. This significantly reduces the time wasted on polling for the write completion, generating a 75% further improvement. Our further profiling reveals that Hermit reduces the per-page overhead of `rmap` by 59% from 1.70 µs to 0.69 µs, TLB shootdown by 92% from 2.45 µs to 0.20 µs, and I/O writes by 88% from 6.47 µs to 0.76 µs. To summarize, Hermit improves the single-thread page reclamation throughput from 77K pages/s to 221K pages/s, making reclamation 2.9× more efficient.

6.4.3 Effectiveness of Feedback-directed Asynchrony

To demonstrate the importance of Hermit's feedback-directed asynchrony, we modified Hermit's reclaim scheduler to use Fastswap's static scheduling policy. The new version `Hermit*` uses a fixed number of reclaim threads and starts reclamation only when the free local memory size falls below 8 MiB. Figure 13 shows the results of Memcached.

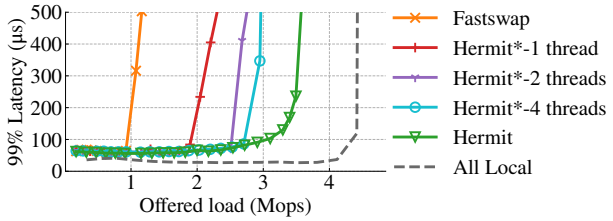


Figure 13: Hermit's feedback-directed asynchrony is indispensable for achieving superior performance. Hermit considerably outperforms all `Hermit*`s—the modified versions that adopt Fastswap's static scheduling policy for reclamation.

Hermit consistently outperforms all variants of `Hermit*`, regardless of the number of reclaim threads statically configured. Our further profiling reveals that the memory pressure during Memcached's execution varies over time. In most cases, it only requires ≤2 reclaim threads to mitigate the pressure. However, upon sudden bursts of requests, it needs up to 4 threads to fully keep up with the demand. Hermit's reclaim scheduler dynamically adjusts the number of reclaim threads to adapt to the changes in demand, thereby achieving superior performance to its static counterparts.

6.4.4 Breaking Down End-to-End Speedup

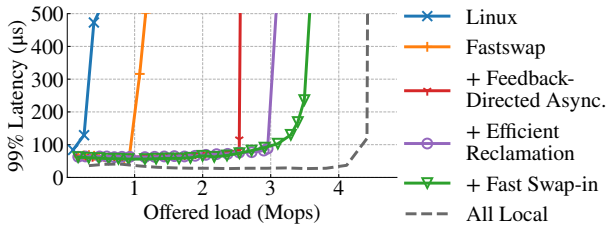


Figure 14: All three of Hermit's optimizations work in tandem to improve Memcached's latency and throughput. Results are measured with 70% local memory.

We evaluated the individual contribution of each of the three optimizations (§6.4.1-§6.4.3) to the overall application performance.

For latency-critical applications, we used Memcached as the representative. We re-ran Memcached with the same configuration as Figure 8 (a) with optimizations enabled incrementally. Figure 14 reports the results. Linux even fails to handle low load of 0.5 Mops under 70% local memory, as it frequently triggers direct reclamation which can easily prolong Memcached's 99th percentile latency by hundreds of microseconds. Fastswap outperforms Linux by offloading reclamation to a dedicated core. However, the application quickly saturates the core's reclamation capacity once the load reaches 1.1 Mops, and starts to trigger direct reclamation again (see Figure 11). This prevents Fastswap from maintaining low 99th percentile latency afterward.

With the reclaim scheduler (§4.2), Hermit can handle a much higher load, 2.5 Mops, before the latency starts to spike. This is because Hermit's reclaim scheduler proactively and timely starts asynchronous reclamation, eliminat-

ing the blocking caused by direct reclamation. Optimizations in the reclamation path (§4.4) reduce the amount of CPU resources required. This alleviates the contention between reclaim threads and application threads, adding 0.4 Mops to the load capacity. Finally, optimizations in the swap-in path (§4.3) make remote memory accesses faster and reduce the per-request processing time, thereby enabling Memcached to achieve higher load with the same amount of compute. Putting them all together, Hermit helps Memcached reach 3.5 Mops using 70% local memory while maintaining 99th percentile latency under 250 µs.

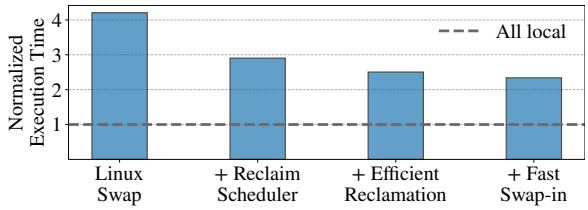


Figure 15: All three of Hermit's optimizations collectively improve Spark's throughput. The Y-axis shows the execution time normalized to the ideal local-only time (68.4s). Results are measured under 20% local memory.

For batch applications, we used Spark as the representative and re-ran it under 20% local memory with the same configuration as Figure 9(a). Figure 15 breaks down the performance improvements. Our reclaim scheduler again improves the application throughput by a large margin (31%) due to the following reasons. First, batch applications usually follow the epochal hypothesis [46], whose compute and memory behaviors vary during an epoch but repeat across epochs. Asynchronous reclamation unleashes the hidden parallelism by speculatively reclaiming pages, making it possible for reclaim threads to efficiently harness idle compute resources in each epoch. Second, Linux swap frequently triggers massive direct reclamations instantaneously, causing severe lock contentions between page faults handlings (swap-in) and reclamation. Hermit avoids the burst of reclamation and greatly alleviates the contention by reclaiming asynchronously and proactively. Further, optimizations on the page reclamation path and the swap-in path collectively improve the swap efficiency: they yield an additional 10% and 4% throughput improvement, respectively.

6.4.5 Resource Consumption of Swap Operations

Network Bandwidth. Hermit performs swap operations eagerly to improve performance. It opportunistically bypasses swap-in deduplication to reduce swap-in latency (§4.3) and proactively schedules asynchronous reclaim threads to avoid direct reclamation (§4.2). These optimizations offer performance benefits potentially at the cost of additional network usage. For example, Hermit might swap in the same page several times in the presence of concurrent page faults. To confirm that Hermit does not incur excessive network traffic, we measure the network bandwidth used for swap-ins and swap-outs, and compare it with Fastswap's usage.

Figure 16 shows the results when running Memcached. The X-axis shows the offered load while the Y-axis shows the average network bandwidth. The error bar quantifies the bandwidth fluctuation during the application's execution. With higher offered load, both Fastswap and Hermit use more network bandwidth as Memcached swaps memory more frequently. The bandwidth usage in swap-outs is lower than in swap-in as clean pages do not need to be written back during reclamation.

For swap-in, Hermit incurs similar network bandwidth usage compared to Fastswap. This is consistent with our further investigation which reveals that the conflict rate (*i.e.* the ratio of concurrent page faults that swap in the same page) is less than 0.07%. Therefore, Hermit's swap-in optimization barely introduces any extra network overhead in practice.

For swap-out, we break down the total bandwidth consumption into the usage of asynchronous swap-out and direct swap-out. Hermit is able to constantly perform asynchronous reclamation without using additional network bandwidth compared to Fastswap. This makes sense as Hermit's optimizations to reclamation timing and efficiency do not inflate the number of reclaimed pages.

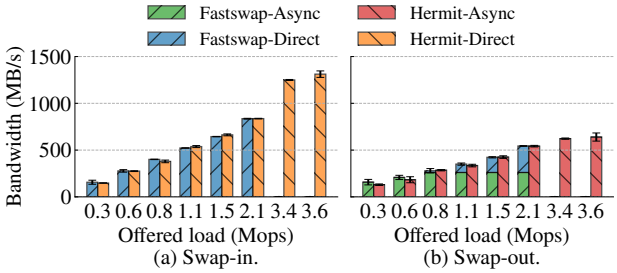


Figure 16: Hermit's optimizations do not incur additional network usage during swap-ins/-outs compared to Fastswap.

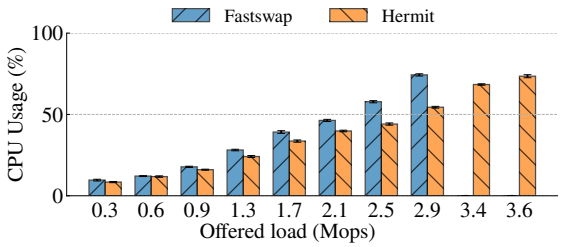


Figure 17: Hermit saves ∼30% CPU cycles under varying load compared with Fastswap, which is the key enabler to achieve low 99th percentile latency under high load.

CPU Cycles. We also profiled the CPU usage of applications running on Fastswap and Hermit, revealing that Hermit can serve much higher load with the same amount of CPU resources. Figure 17 depicts the total CPU usage of Memcached and Hermit's reclaim threads under 70% local memory ratio and varying load. When increasing load, both Fastswap and Hermit use more CPU cycles as Memcached

swaps more frequently. We observed that Memcached fails to use > 70% CPU cycles due to its internal lock contention on hot slabs under skewed workloads. Even though Hermit can spawn more reclaim threads than Fastswap (when needed), it uses 20%–30% fewer CPU resources overall, thanks to its feedback-directed asynchrony and more effective use of batching. Therefore, Hermit is able to offer 32% higher load capacity for Memcached compared to Fastswap.

7 Related Work

Resource Disaggregation. Datacenters today suffer from poor average resource utilization due to overprovisioning [41, 56]. Resource disaggregation, an idea that dates back to 1990s [12, 23, 31, 45, 61], has gained renewed interest, thanks to the high performance of modern datacenter networks [16, 19, 28]. Its key idea is to break the server boundary and unstrand idle resources of remote servers over the network. Existing systems have demonstrated the feasibility of disaggregating various types of resources, including storage [32, 35, 37], accelerators [6, 20], and memory [10, 43, 52, 57]. Some other systems focus on improving the reliability of disaggregated datacenters [33, 64]. We focus on memory disaggregation (*i.e.*, remote memory) in this paper.

Kernel-based Remote Memory. To provide transparency to existing applications, the kernel-based approach leverages OS paging to access and manage remote memory. Most kernel-based systems build upon Linux, including Hermit. Infiniswap [29] is an early work that integrates Linux's swap subsystem with an RDMA-based block device backend. Later, Fastswap [10] leverages the lightweight `frontswap` interface of Linux to reduce overhead and offloads page reclamation to a dedicated core. Leap [43] improves Linux's prefetcher to achieve a higher local memory hit ratio. Canvas [59] isolates swap paths for co-running applications. The ongoing advances of Linux's virtual memory subsystem from the kernel community also benefit Linux-based remote memory. These include, but are not limited to multi-generational LRU [55], speculative page faults [4], maple-tree-based VMAs [2], and DAMON-based proactive page reclamation [5]. Finally, LegoOS [53] makes larger changes to both the kernel and hardware with the goal of achieving better performance through a clean-slate approach.

Library-based Remote Memory. Library-based approaches bypasses the OS to reduce kernel overhead and overcome the granularity restrictions imposed by paging. They trade application transparency for performance; application developers often have to modify their code to use new remote memory APIs. FaRM [25] and KVDirect [38] expose remote memory with an external key-value store interface which mismatches with the construction of existing applications. Distributed shared memory (*e.g.*, [40, 44]), on the other hand, provides an object-oriented interface that is more user-friendly. AIFM [52] proposes a higher-level abstraction of remote-able data structure, but but it still requires effort to port applications. Semeru [57], Mako [42], and MemLiner [58] are JVM-based remote memory runtimes, offering transparency to Java applications by co-designing the JVM with the kernel.

Hardware-accelerated Remote Memory. Another type of work proposes novel hardware designs, thereby unlocking new opportunities for optimizing remote memory. While Hermit focuses on the software layer, it could benefit from advances to the underlying hardware. PBerry [18] and Kona [17] overcome the granularity restriction of paging and enable cache-line-level remote memory access. Clio [30], StRoM [54], and RMC [11] reduce the expensive network traffic by offloading tasks into the customized hardware of the memory server. Finally, the emerging CXL bus [39] may lower the performance cost of accessing remote memory by delivering lower latency and near-local-DRAM throughput.

Multi-tiered Memory System. Recent research has focused on overcoming DRAM's capacity wall through the use of slower memory/storage devices—such as compressed memory, non-volatile memory (NVM), NVMe SSD, *etc.* Two examples of such systems are TMO [60] and HeMem [50], which transparently offload main memory to slower tiers. TMO focuses on developing a policy for determining which data to offload and how much, whereas Hermit aims at building an efficient offloading mechanism. HeMem targets improving throughput for batch applications. Therefore, it treats page offloading as a time-insensitive operation and performs it in the background. In contrast, Hermit optimizes for both batch and latency-critical applications by conducting page reclamation timely and proactively.

8 Conclusion

In this paper, we presented Hermit, a re-architected swap system that is based on adaptive, feedback-directed asynchrony. Our evaluation shows that Hermit significantly outperforms Fastswap (the state-of-the-art swap system) in real data center applications; it reduces the 99th percentile tail latency by 99.7% and improves the throughput by 1.24× on average. Hermit defies the conventional wisdom about kernel-based remote memory, demonstrating that it is possible to achieve both full transparency and high performance simultaneously.

Acknowledgements

We thank the anonymous reviewers for their valuable and thorough comments. We are grateful to our shepherd Michael Wei for his feedback. This work is supported by NSF grants CNS-1703598, CCF-1723773, CNS-1763172, CCF-1764077, CNS-1907352, CHS-1956322, CNS-2007737, CNS-2006437, CNS-2128653, CCF-2106404, CNS-2106838, CNS-2147909, CNS-2104398, ONR grant N00014-18-1-2037, research grants from Cisco, Intel CAPA, VMware, and Samsung, and a gift from Amazon.

References

[1] gdnsd - an authoritative-only dns server. https://gdnsd.org/.

[2] Introducing maple trees. https://lwn.net/Articles/845507/.

[3] NVMe over fabrics. http://community.mellanox.com/s/article/what-is-nvme-over-fabrics-x.

[4] Speculative page faults. https://lwn.net/Articles/851853/.

[5] Using damon for proactive reclaim. https://lwn.net/Articles/863753/.

[6] Virtual gpu (vgpu) | nvidia. https://www.nvidia.com/en-us/data-center/virtual-solutions/.

[7] Memcached - a distributed memory object caching system. http://memcached.org, 2020.

[8] Wikipedia networks data. http://konect.uni-koblenz.de/networks/, 2020.

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

[10] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker. Can far memory improve job throughput? In *EuroSys*, 2020.

[11] E. Amaro, Z. Luo, A. Ousterhout, A. Krishnamurthy, A. Panda, S. Ratnasamy, and S. Shenker. Remote memory calls. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, HotNets '20, pages 38–44, New York, NY, USA, 2020. Association for Computing Machinery.

[12] T. Anderson, D. Culler, and D. Patterson. A case for now (networks of workstations). *IEEE Micro*, 15(1):54–64, 1995.

[13] D. Ardelean, A. Diwan, and C. Erdman. Performance analysis of cloud applications. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 405–417, Renton, WA, Apr. 2018. USENIX Association.

[14] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[15] P. Baldi, P. Sadowski, and D. Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature communications*, 5(1):1–9, 2014.

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

[17] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli. *Rethinking Software Runtimes for Disaggregated Memory*, pages 79–92. Association for Computing Machinery, New York, NY, USA, 2021.

[18] I. Calciu, I. Puddu, A. Kolli, A. Nowatzyk, J. Gandhi, O. Mutlu, and P. Subrahmanyam. Project pberry: Fpga acceleration for remote memory. HotOS '19, pages 127–135, New York, NY, USA, 2019. Association for Computing Machinery.

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

[20] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. A cloud-scale acceleration architecture. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49. IEEE Press, 2016.

[21] T. Chen and C. Guestrin. extreme gradient boosting for applied machine learning. https://xgboost.readthedocs.io/en/latest/, 2021.

[22] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[23] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94, pages 19–es, USA, 1994. USENIX Association.

[24] C. Delimitrou and C. Kozyrakis. Amdahl's law for tail latency. *Commun. ACM*, 61(8):65–72, jul 2018.

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

[26] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.

[27] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[28] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network requirements for resource disaggregation. In *OSDI*, pages 249–264, 2016.

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

[30] Z. Guo, Y. Shan, X. Luo, Y. Huang, and Y. Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2022, pages 417–433, New York, NY, USA, 2022. Association for Computing Machinery.

[31] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. *Commun. ACM*, 53(10):85–93, oct 2010.

[32] J. Hwang, Q. Cai, A. Tang, and R. Agarwal. TCP ≈ RDMA: CPU-efficient remote storage access with i10. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 127–140, Santa Clara, CA, Feb. 2020. USENIX Association.

[33] S. Kadekodi, F. Maturana, S. Athlur, A. Merchant, K. V. Rashmi, and G. R. Ganger. Tiger: Disk-Adaptive redundancy without placement restrictions. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 413–429, Carlsbad, CA, July 2022. USENIX Association.

[34] A. Kalia, M. Kaminsky, and D. G. Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *OSDI*, pages 185–201, 2016.

[35] A. Klimovic, H. Litz, and C. Kozyrakis. ReFlex: Remote flash ≈ local flash. In *ASPLOS*, pages 345–359, 2017.

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

[37] S. Legtchenko, H. Williams, K. Razavi, A. Donnelly, R. Black, A. Douglas, N. Cheriere, D. Fryer, K. Mast, A. D. Brown, A. Klimovic, A. Slowey, and A. Rowstron. Understanding rack-scale disaggregated storage. In *HotStorage*, 2017.

[38] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 137–152, New York, NY, USA, 2017. Association for Computing Machinery.

[39] H. Li, D. S. Berger, S. Novakovic, L. Hsu, D. Ernst, P. Zardoshti, M. Shah, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini. First-generation memory disaggregation for cloud platforms, 2022.

[40] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, Nov. 1989.

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

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

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

[44] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-tolerant software distributed shared memory. In *USENIX ATC*, pages 291–305, 2015.

[45] T. Newhall, S. Finney, K. Ganchev, and M. Spiegel. Nswap: A network swapping module for linux clusters. In *European Conference on Parallel Processing*, 2003.

[46] K. Nguyen, L. Fang, G. Xu, B. Demsky, S. Lu, S. Alamian, and O. Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *OSDI*, pages 349–365, 2016.

[47] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *NSDI*, pages 361–378, 2019.

[48] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *OSDI*, pages 1–16, 2014.

[49] A. Raybuck, T. Stamler, W. Zhang, M. Erez, and S. Peter. Hemem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 392–407, New York, NY, USA, 2021. Association for Computing Machinery.

[50] A. Raybuck, T. Stamler, W. Zhang, M. Erez, and S. Peter. Hemem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 392–407, New York, NY, USA, 2021. Association for Computing Machinery.

[51] R. A. Rossi and N. K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[52] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay. AIFM: High-performance, application-integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 315–332. USENIX Association, Nov. 2020.

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

[54] D. Sidler, Z. Wang, M. Chiosa, A. Kulkarni, and G. Alonso. Strom: Smart remote memory. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[55] The OCP Foundation. Multi-generational lru: the next generation. https://lwn.net/Articles/856931/.

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

[57] C. Wang, H. Ma, S. Liu, Y. Li, Z. Ruan, K. Nguyen, M. D. Bond, R. Netravali, M. Kim, and G. H. Xu. Semeru: A memory-disaggregated managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 261–280. USENIX Association, Nov. 2020.

[58] C. Wang, H. Ma, S. Liu, Y. Qiao, J. Eyolfson, C. Navasca, S. Lu, and G. H. Xu. MemLiner: Lining up tracing and application for a far-memory-friendly runtime. In *OSDI*, 2022.

[59] C. Wang, Y. Qiao, H. Ma, S. Liu, Y. Zhang, W. Chen, R. Netravali, M. Kim, and G. H. Xu. Canvas: Isolated and adaptive swapping for multi-applications on remote memory. In *NSDI*, 2023.

[60] J. Weiner, N. Agarwal, D. Schatzberg, L. Yang, H. Wang, B. Sanouillet, B. Sharma, T. Heo, M. Jain, C. Tang, and D. Skarlatos. Tmo: Transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, pages 609–621, New York, NY, USA, 2022. Association for Computing Machinery.

[61] D. Williams, H. Jamjoom, Y.-H. Liu, and H. Weatherspoon. Overdriver: Handling memory overload in an oversubscribed cloud. *SIGPLAN Not.*, 46(7):205–216, Mar 2011.

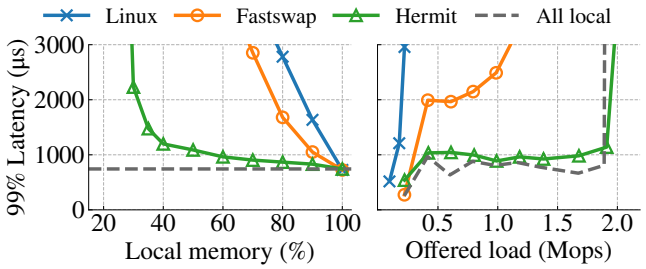
[62] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. HotCloud, page 10, Berkeley, CA, USA, 2010.

[63] I. Zhang, A. Raybuck, P. Patel, K. Olynyk, J. Nelson, O. S. N. Leija, A. Martinez, J. Liu, A. K. Simpson, S. Jayakar, P. H. Penna, M. Demoulin, P. Choudhury, and A. Badam. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *SOSP*, pages 195–211, 2021.

[64] Y. Zhou, H. M. G. Wassel, S. Liu, J. Gao, J. Mickens, M. Yu, C. Kennelly, P. Turner, D. E. Culler, H. M. Levy, and A. Vahdat. Carbink: Fault-Tolerant far memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 55–71, Carlsbad, CA, July 2022. USENIX Association.

A Tail Latency of Linux-Based Applications

In this section, we evaluate Hermit using the vanilla Linux-based Memcached as opposed to the Shenango-enhanced Memcached (§6.2). Figure 18(a) shows 99th percentile latency with fixed load (1 Mops) and varying local memory ratios. Figure 18(b) shows latency with a fixed local memory ratio (70%) and varying load. Hermit still significantly outperforms other baseline systems. The results show a similar trend to the results of Shenango-enhanced Memcached.



(a) Under 1 Mops load. (b) Under 70% local memory.

Figure 18: For the vanilla Linux-based Memcached, Hermit still significantly outperforms other baseline systems.

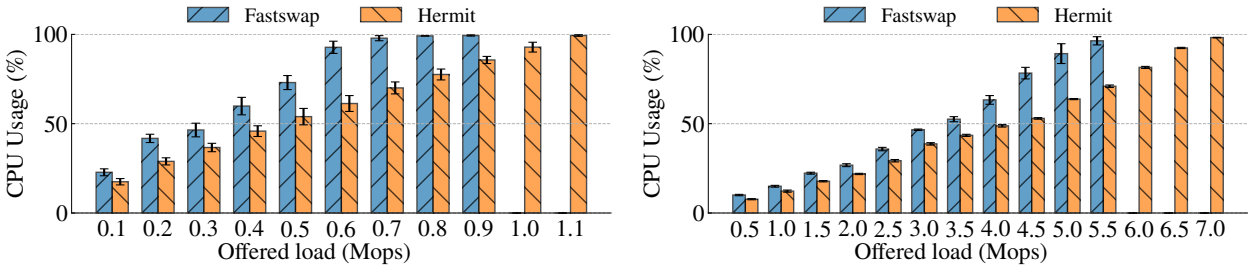
B CPU Usage of Other Applications

This section reports and compares the CPU usage of SocialNet and Gdnsd running on Fastswap and Hermit under the same setting as in Figure 8. Figure 19 shows the results. Thanks to its efficient swap design, Hermit consistently uses 10%–40% fewer CPU cycles than Fastswap, even though it invokes more reclaim threads.

C Tail Latency in Other Percentiles

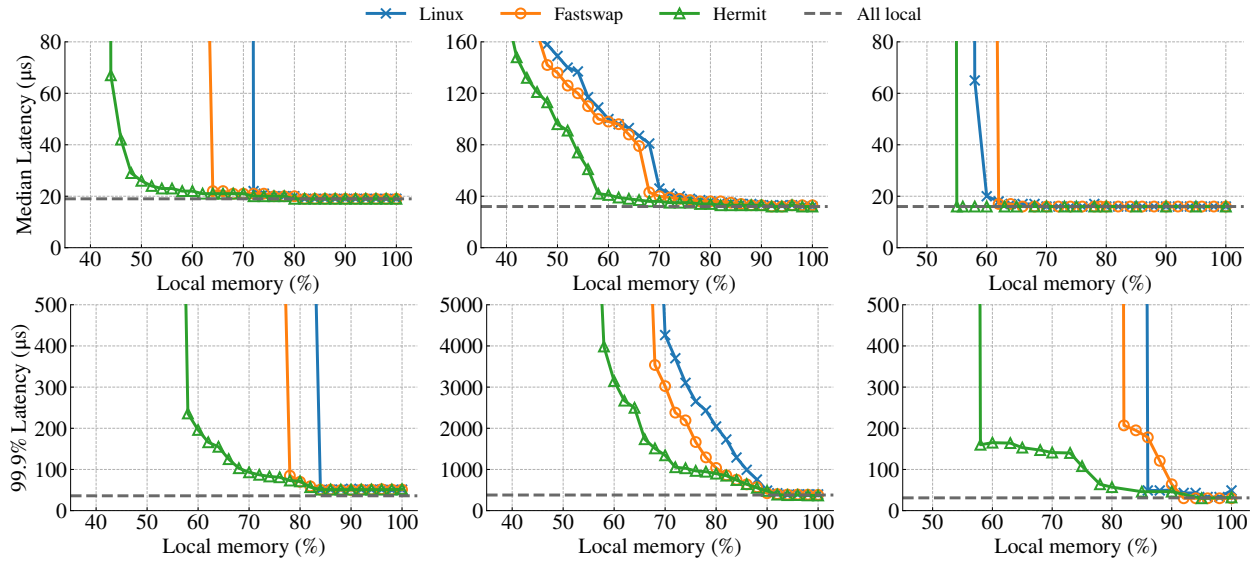
This section reports the median and 99.9th percentile latency of all three latency-critical applications. Figure 20 depicts the results when running applications with a fixed load and varying local memory ratios. The results exhibit a similar trend to Figure 7. All three systems sustain low median latency when local memory is not too scarce, while Hermit slightly outperforms Fastswap and Linux. When local memory continues to decrease, applications have to spend more CPU cycles on frequent remote memory accesses. The CPU congestion consequently ramps up the median latency. Thanks to the CPU-efficient swap design, Hermit's median latency increases slower than Fastswap, allowing applications to serve higher load, particularly when local memory is scarce. With regards to 99.9th percentile latency, Hermit again significantly outperforms Fastswap and Linux. It enables applications to put on average 20% more working set in remote memory without violating the tail latency agreement.

Next, we repeated the experiment shown in Figure 8 by fixing the local memory ratio to 70% and measured the median and 99.9th percentile latency of applications with varying load (see Figure 21). Hermit is able to deliver low median latency close to the ideal setup and much lower tail latency. Since 99.9th percentile latency is more susceptible to direct page reclamation, Fastswap experiences significant performance degradation once its single dedicated core gets saturated. Hermit, in contrast, is able to offer high load (> 60% compared to the ideal all-local setup) and maintain low 99.9th percentile latency under 70% local memory ratio.



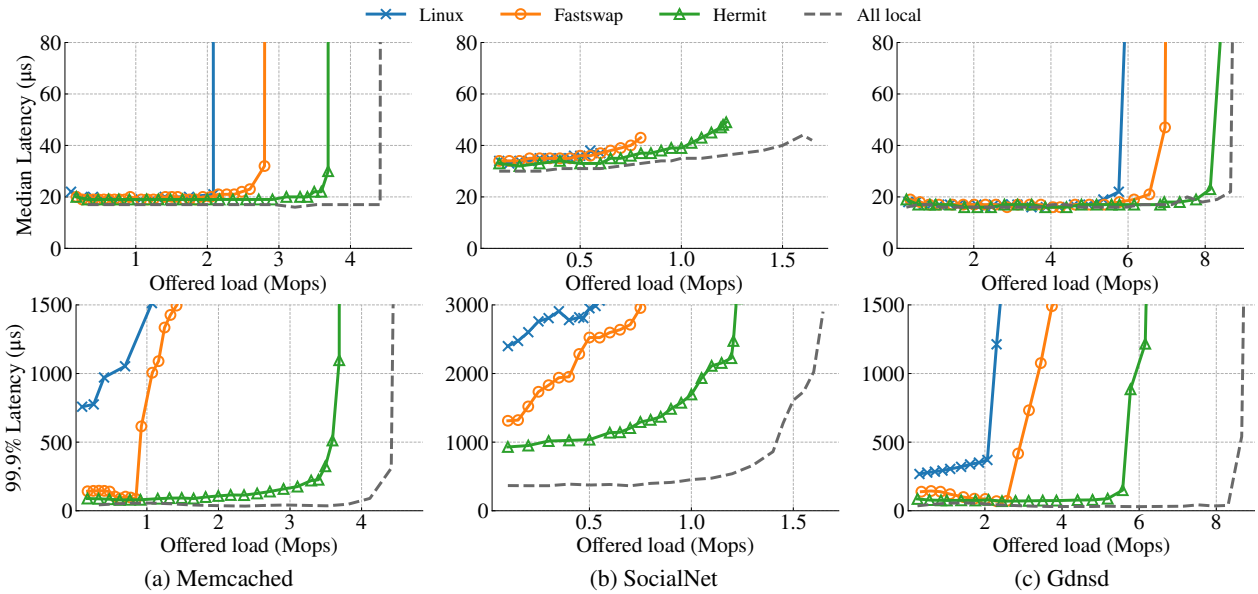
(a) SocialNet (b) Gdnsd

Figure 19: For SocialNet and Gdnsd, Hermit still saves 10%–40% CPU cycles under varying load compared with Fastswap, which is the key enabler it can achieve low tail latency under high load.



(a) Memcached (1.5 Mops) (b) SocialNet (0.75 Mops) (c) Gdnsd (4 Mops)

Figure 20: Hermit still significantly outperforms Fastswap and Linux in terms of median and 99.9% latency under the same load and varying local memory ratio.



(a) Memcached (b) SocialNet (c) Gdnsd

Figure 21: Hermit also achieves significantly lower median and 99.9% latency than Fastswap and Linux under 70% local memory ratio and varying load. As we used a closed-loop load generator for SocialNet, it reaches the maximum load capacity before its median latency spikes.