Canvas: Isolated and Adaptive Swapping for Multi-Applications on Remote Memory

Chenxi Wang[†*] Yifan Qiao[†*] Haoran Ma[†] Shi Liu[†] Yiying Zhang[‡]
Wenguang Chen[§] Ravi Netravali[♯] Miryung Kim[†] Guoqing Harry Xu[†]
UCLA[†] *UCSD*[‡] *Tsinghua University*[§] *Princeton University*[♯]

Abstract

Remote memory techniques for datacenter applications have recently gained a great deal of popularity. Existing remote memory techniques focus on the efficiency of a single application setting only. However, when multiple applications co-run on a remote-memory system, significant interference could occur, resulting in unexpected slowdowns even if the same amounts of physical resources are granted to each application. This slowdown stems from massive sharing in applications' swap data paths. Canvas is a redesigned swap system that fully isolates swap paths for remote-memory applications. Canvas allows each application to possess its dedicated swap partition, swap cache, prefetcher, and RDMA bandwidth. Swap isolation lays a foundation for adaptive optimization techniques based on each application's own access patterns and needs. We develop three such techniques: (1) adaptive swap entry allocation, (2) semantics-aware prefetching, and (3) two-dimensional RDMA scheduling. A thorough evaluation with a set of widely-deployed applications demonstrates that Canvas minimizes performance variation and dramatically reduces performance degradation.

1 Introduction

Techniques enabling datacenter applications to use far memory [36, 39, 8, 62, 73, 91, 104, 90, 19] have gained traction due to their potential to break servers' memory capacity wall, thereby improving performance and resource utilization. Existing far-memory techniques can be roughly classified into two categories: (1) clean-slate techniques [90, 19] that provide new primitives for developers to manage remote memory, and (2) swap-based techniques [39, 91, 8, 104, 2] that piggyback on existing swap mechanisms in the OS kernel. Clean-slate techniques provide greater efficiency by enabling user-space far memory accesses, while swap-based techniques offer transparency, allowing legacy code to run *as is* on a far-memory system. This paper focuses on swap mechanisms as they are more practical and easier to adopt.

A typical swap system in the OS uses a *swap partition* and *swap cache* for applications to swap data between memory and external storage. The swap partition is a storage-backed swap space. The swap cache is an intermediate buffer between the *local memory* and storage—it caches *unmapped pages* that were just swapped in or are about to be swapped out. Upon a page fault, the OS looks up the swap cache; a cache miss would trigger a *demand swap* and a number of *prefetching swaps*. Swaps are served by RDMA and all fetched pages are initially placed in the swap cache. The demand page is then mapped to a virtual page and moved out of the swap cache, completing the fault handling process.

Problems. Current swap systems run multiple applications over shared swap resources (*i.e.*, swap partition, RDMA, *etc.*). This design works for *disk-based swapping* where disk access is slow—each application can allow only a tiny number of pages to be swapped to maintain an acceptable overhead. This assumption, however, no longer holds under far memory because an application can place more data in far memory than local memory and yet still be efficient, thanks to RDMA's low latency and high bandwidth.

As such, applications have orders-of-magnitude more swap requests under far memory than disks. Millions of swap requests from different applications go through the same shared data path in a short period of time, leading to *severe performance interference*. Our experiments show that, with the same amounts of CPU and local-memory resources, co-running applications leads up to a 6× slowdown, an overhead unacceptable for any real-world deployment.

State of the Art. Interference is a known problem in datacenter applications and a large body of work exists on isolation of CPU [64, 16, 25], I/O [40, 96], network bandwidth [13, 37, 94, 87, 77, 53] and processing [59]. Most of these techniques build on Linux's `cgroup` mechanism, which focuses on isolation of traditional resources such as CPU and memory, *not* swap resources such as remote memory usage and RDMA. Prior swap optimizations such as Infiniswap [39] and Fastswap [8] focus on reducing remote access latency, overlooking the impact of swap interference in realistic settings. Justitia [113] isolates RDMA bandwidth between applications, but does not eliminate other types of interference such as locking and swap cache usage.

Contribution #1: Interference Study (§3). We conducted a systematic study with a set of widely-deployed applications on Linux 5.5, the latest kernel version compatible with Mellanox's latest driver (4.9-3.1.5.0) for our InfiniBand card. Our results reveal three major performance problems:

- **Severe lock contention:** Since all applications share a single swap partition, extensive locking is needed for swap entry allocation (needed by every swap-out), reduc-

∗ Contributed equally.

ing throughput and precluding full utilization of RDMA's bandwidth. Our experience shows that in windows of frequent remote accesses, applications can spend **70%** of the windows' time on swap entry allocation.

- **Uncontrolled use of swap resources (*e.g.*, RDMA):** The use of the shared RDMA bandwidth is often dominated by the pages fetched for applications with many threads simultaneously performing frequent remote accesses. For example, aggressively (pre)fetching pages to fulfill one application's needs can disproportionally reduce other applications' bandwidth usage. Further, even within one application, prefetching competes for resources with demand swaps, leading to either prolonged fault handling or delayed prefetching that fails to bring back pages in time.
- **Reduced prefetching effectiveness:** Applications use the same prefetcher, prefetching data based on *low-level (sequential or strided) access patterns* across applications. However, modern applications exhibit far more diverse access patterns, making it hard for prefetching to be effective across the board. For example, co-running Spark and native applications reduces Leap [73]'s prefetching contribution by **3.19**×.

These results highlight two main problems. First, interference is caused by sharing a combination of swap resources including the swap partition/cache, and RDMA (bandwidth and SRAM on RNIC). Although recent kernel versions added support [47] for charging prefetched pages into `cgroup`, resolving interference requires a *holistic* approach that can isolate all these resources. Furthermore, interference stems not only from resource racing, but also from fundamental limitations with the current design of the swap system. For instance, reducing interference between prefetching and demand swapping requires understanding whether a prefetching request can come back in time. If not, it should be dropped to give resources to demand requests, which are on the critical path. This, in turn, requires a redesign of the kernel's fault handling logic.

Second, cloud applications exhibit highly diverse behaviors and resource profiles. For example, applications with a great number of threads are more sensitive to locking than single-threaded applications. Furthermore, managed applications such as Spark often make heavy use of reference-based data structures while native applications are often dominated by large arrays. The *application-agnostic nature* of the swap system makes it hard for a one-size-fits-all policy (*e.g.*, a global prefetcher) to work well for diverse applications. Effective per-application policies dictates (1) holistic swap isolation and (2) understanding application semantics, which is currently inaccessible in the kernel.

Contribution #2: Holistic Swap Isolation (§4). To solve the first problem, we develop Canvas, a *fully-isolated* swap system, which enables each application to have its dedicated swap partition, swap cache, and RDMA usage. In doing so, Canvas can charge each application's `cgroup` for the usage of all kinds of swap resources, preventing certain applications from aggressively invading others' resources.

Contribution #3: Isolation-Enabled Adaptive Optimizations (§5). To solve the second problem, we develop a set of adaptive optimizations that can tailor their policies and strategies to application-specific swap behaviors and resource needs. Our adaptive optimizations bring a *further boost* on top of the isolation-provided benefits, making co-running applications even *outperform* their individual runs.

(1) Adaptive Swap Entry Allocation (§5.1) Separating swap partitions reduces lock contention at swap entry allocations to a certain degree, but the contention can still be heavy for multi-threaded applications. For example, Spark creates many threads to fully utilize cores and these threads need synchronizations before obtaining swap entries. The synchronization overhead increases dramatically with the number of cores (§6.4.1), creating a scalability bottleneck. We develop an adaptive swap entry allocator that dynamically balances between the degree of lock contention (*i.e.*, time) and the amount of swap space needed (*i.e.*, space) based on each application's memory behaviors.

(2) Adaptive Two-tier Prefetching (§5.2) Current kernel prefetchers build on low-level access patterns (*e.g.*, sequential or strided). Although such patterns are useful for applications with large array usages, many cloud applications are written in high-level, managed languages such as Java or Python; their accesses come from multiple threads or exhibit pointer-chasing behavior as opposed to sequential or strided patterns. As effective prefetching is paramount to remote-memory performance, Canvas employs a two-tier prefetching design. Our *kernel-tier prefetcher* prefetches data for each application into its private swap cache based on low-level patterns. Once this prefetcher cannot effectively prefetch data, Canvas adaptively forwards the faulty address up to the *application tier* via a modified `userfaultfd` interface, enabling customized prefetching logic at the level of reference-based or thread-based access patterns.

(3) Adaptive RDMA Scheduling (§5.3) Isolating RDMA bandwidth alone for each application is insufficient. As there could be many more *prefetching* requests than *demand swap requests*, naïvely sending all to RDMA delays demand requests, increasing fault-handling latency. On the other hand, naïvely delaying prefetching requests (as in FastSwap [8]) reduces their *timeliness*, making prefetched pages useless. We built a *two-dimensional* RDMA scheduler, which schedules packets not only between applications but also between prefetching and demand requests for each application.

Results. Our evaluation (§6) with a set of 14 widely-deployed applications (including Spark [109], Cassandra [10], Neo4j [79], Memcached [4], XGBoost [23, 22], Snappy [38], *etc.*) demonstrates that Canvas improves the overall application performance by up to **6.2**× (average **3.5**×) and reduces applications' performance variation (*i.e.*,

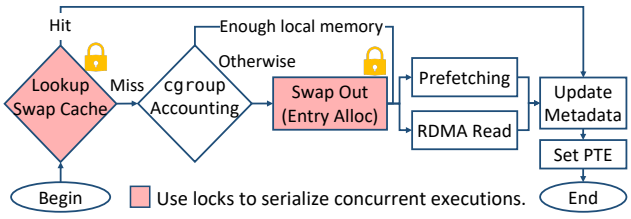


Figure 1: The kernel's remote-access data path.

standard deviation) by **7**×, from an overall of **1.72** to **0.23**. Canvas improves the overall RDMA bandwidth utilization by **2.8**× for co-run applications. Canvas is available at https://github.com/uclasystem/canvas.

2 Background

This section presents the necessary background in Linux 5.5, which is the latest kernel version compatible with Mellanox's latest driver for our InfiniBand adapter.

Figure 1 illustrates the kernel's remote access data path where remote memory is mapped into the host server as a swap partition where applications access remote memory via page faults. The swap partition is split into a set of 4KB *swap entries*, each mapping to an actual remote memory cell and has a unique entry ID. Upon a page fault, the kernel uses the swap entry ID contained in the corresponding page table entry (PTE) to locate the swap entry that stores the page.

The first step in handling the fault is to look up the swap cache, which is a set of radix trees, each containing a number of cached and unmapped pages for a block (*e.g.*, 64MB) of swap entries. These pages were either just swapped in due to demand swapping or prefetching, or are about to be swapped out. If a page can be found there, it gets mapped to a virtual page and removed from the swap cache. Otherwise, the kernel needs to perform a *demand swap-in*.

Before issuing the request, the kernel first does `cgroup` accounting to understand if there is enough physical memory to swap in the page. If there is, the kernel issues an RDMA read request, which is then pushed into RDMA's dispatch queue. As the demand swap occurs, the kernel prefetches a number of pages that will likely be needed in the future. This number depends on the swap history at the past few page faults. For example, if the pages fetched follow a sequential or strided pattern, the kernel will use this pattern to fetch a few more pages. If no pattern is found, the kernel reduces the number of prefetched pages until it stops prefetching completely. Once these demand and prefetched pages arrive, they are placed into the swap cache. Their swap entries in remote memory are then freed.

If `cgroup` accounting deems that local memory is insufficient for the new page, the kernel uses an LRU algorithm to evict pages. Evicting a page *unmaps* it and pushes it into the swap cache. When memory runs low, the kernel releases existing pages from the swap cache to make room for newly fetched pages. Clean pages can be removed right away and dirty pages must be written back. To write back a page, the swap system must first allocate a swap entry using a free-list-based allocation algorithm. Finally, an RDMA write request is generated and the page is written into the entry via RDMA.

In each remote access, extensive locking is needed for swap entry allocation—shared allocation metadata (*e.g.*, free list) must be protected when multiple applications/threads request swap entries simultaneously. Although there are active efforts [48, 46] in the Linux community to optimize swap entry allocation, their performance and scalability is unsatisfactory for cloud workloads (see Appendix B).

3 Motivating Performance Study

To understand the impact of interference, we conducted a study with a set of widely-deployed applications including Apache Spark [109], Neo4j [79], XGBoost [23] (*i.e.*, a popular ML library), Snappy [38] (*i.e.*, Google's fast compressor/decompressor), as well as Memcached [4]. Spark and Neo4j are managed applications running on the JVM, while the other three are native applications. They cover a spectrum of cloud workloads from data storage through analytics to ML. In addition, they include both batch jobs (such as Spark) and latency-sensitive jobs (such as Memcached). Co-running them represents a typical scenario in a modern datacenter where operators fill left-over cores unused by latency-sensitive tasks with batch-processing applications to improve CPU utilization [15]. For example, in a Microsoft Bing cluster, batch jobs are colocated with latency-sensitive services on over 90,000 servers [49]. Google also reported that 60% of machines in their compute cluster co-run at least five jobs [112].

We ran these programs, individually vs. together, on a machine with two Xeon(R) Gold 6252 processors, running Linux 5.5. Another machine with two Xeon(R) CPU E5-2640 v3 processors and 128GB memory was used for remote memory. Each machine was equipped with a 40 Gbps Mellanox ConnectX-3 InfiniBand adapter and interconnected by one Mellanox 100 Gbps InfiniBand switch. Using `cgroup`, the same amounts of CPU and local memory resources were given to each application throughout the experiments. RDMA bandwidth was *not* saturated for both application individual runs and co-runs. The amount of local memory configured for each application was 25% of its working set.

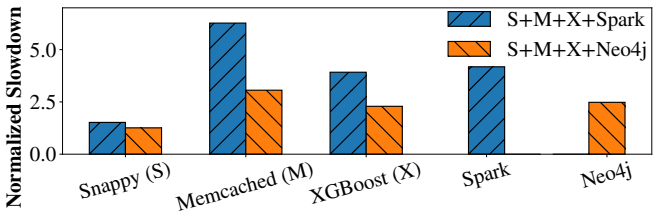


Figure 2: Slowdowns of co-running applications compared to running each individually.

Performance Interference and Degradation. To understand the overall performance degradation and how it changes with different applications, we used two managed applications: Spark and Neo4j. Figure 2 reports each application's performance degradation when co-running with other applications compared to running alone. The blue/orange bars show the slowdowns when the three native applications co-run with Spark/Neo4j. Clearly, co-running applications significantly reduces each application's performance. We observed an overall **3.9/2.2**× slowdown when native applications co-run with Spark/Neo4j. Spark persists a large RDD in memory and keeps swapping in/out different parts of the RDD, while Neo4j is a graph database and holds much of its graph data in local memory and thus does not swap as much as Spark.

Another observation is that the impact of interference differs significantly for different applications. Applications that generate high swap throughputs aggressively invade swap and RDMA resources of other applications. In our experiments, Memcached, XGBoost, and Spark all need frequent swaps. However, Spark runs many more threads (>90 application and runtime threads) than Memcached (4) and XGBoost (16), resulting in a much higher swap throughput. As such, Spark takes disproportionally more resources, leading to severe degradation for Memcached and XGBoost.

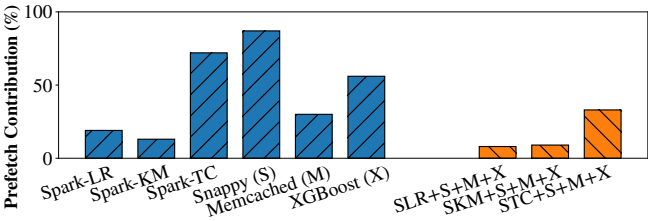
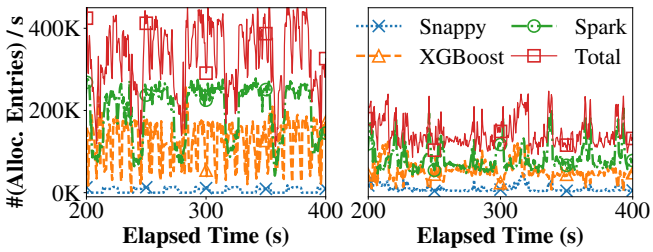


Figure 3: Prefetching contribution of Leap: the percentage of page faults served by Leap-prefetched pages (%).

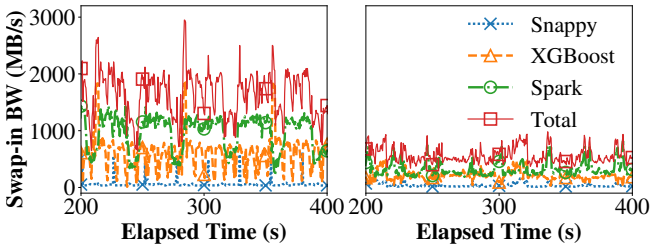
Reduced Prefetching Effectiveness. Sharing the same prefetching policy reduces the prefetching effectiveness when multiple applications co-run. Figure 3 reports *prefetching contribution*—the percentage of page faults served by prefetched pages—the higher the better; if a prefetched page is never used, prefetching it would only incur overhead. We used Leap [73] as our prefetcher. The left six bars report such percentages for the applications running individually. When applications co-run, the rightmost three bars report the average percentages across applications. As shown, co-running dramatically reduces the contribution.

Note that Leap [73] uses a majority-vote algorithm to identify patterns across multiple applications. However, when applications that exhibit drastically different behaviors co-run, Leap cannot adapt its prefetching mechanism and policy to each application. Furthermore, Leap is an aggressive prefetcher—even if Leap does not find any pattern, it always prefetches a number of contiguous pages. However, aggressive prefetching for applications such as Spark with garbage collection (GC) is ineffective—*e.g.*, prefetching for a GC thread has zero benefit and only incurs overhead. Detailed evaluation of prefetching can be found in §6.4.



(a) Running individually. (b) Co-running.

Figure 4: Swap entry allocation throughput when applications run individually (a) and together (b).



(a) Running individually. (b) Co-running.

Figure 5: RDMA swap-in bandwidth when applications run individually (a) and together (b).

Lock Contention. We observed severe lock contention in the swap system when applications co-run, particularly at swap entry allocation associated with each swap-out.

We experimented with Spark (Logistic Regression), XGBoost, and Snappy. Our results show that in windows of frequent remote accesses, co-running applications can spend up to **70%** of the window time on obtaining swap entries. Lock contention leads to significantly reduced swap-entry allocation throughput, reported in Figure 4. The total lines in Figure 4(a) and (b) show the total throughput (*i.e.*, the sum of each application's allocation throughput). The co-running throughput (b) is drastically reduced compared to the individual run's throughput (a) (*i.e.*, ∼450Kps to ∼200Kps).

Reduced RDMA Utilization. Figure 5 compares the RDMA read bandwidth (for swap-ins) when applications run individually and together. Similarly, the total line represents the sum of each application's RDMA bandwidth. The total RDMA utilization is constantly below ∼1000MBps in Figure 5(b), which is **3.28**× lower than that in Figure 5(a) due to various issues (*e.g.*, locking, reduced prefetching, *etc.*). The RDMA write bandwidth degrades by an overall of **2.80**×.

Demand v.s. Prefetching Interference. Optimizations such as Fastswap [8] improve swap performance by dividing the RDMA queue pairs (QP) into sync and async. The high-priority synchronous QP is used for demand swaps, while the low priority async QP is used for prefetching requests.

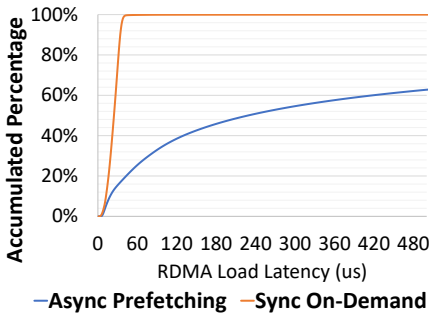


Figure 6: Latency of prefetching and on-demand swapping.

This separation reduces head-of-line blocking incurred by prefetching. However, when applications co-run, this design adds a delay for prefetching. Figure 6 depicts the CDF of the latency of RDMA packets from demand and prefetching requests, when the four applications co-run on Leap. As shown, 99% of the on-demand requests are served within 40μs. However, the latency of 36.9% of prefetching requests is longer than 512μs and it can reach up to 52ms! Long latency renders prefetched pages useless because prefetching is meant to load pages to be used soon. Our profiling shows that among the prefetched pages that are actually accessed by the application, *90% are accessed within 70*μs, indicating that ∼70% of the pages prefetched return too late. A late prefetch of a page would subsequently block a demand request of the page when it is accessed by the application. This problem motivates our two-dimensional RDMA scheduling (§5.3).

Takeaway. The root cause of performance degradation is that multiple applications, whose resource needs and swap behaviors are widely apart, all run on a global swap system with the same allocator and prefetcher. Table 1 summarizes these problems, their performance impact, and our solutions.

4 Swap System Isolation

Canvas extends `cgroup` for users to specify size constraints for swap partition, swap cache, and RDMA bandwidth. We discuss the kernel support to enforce these new constraints, laying a foundation for adaptive optimizations in §5.

Swap Partition Isolation. In Linux, remote memory is managed via a swap partition interface, shared by all applications. If there are multiple available swap partitions, they are used in a *sequential manner* according to their priorities. As a result, data of different applications are mixed and stored in arbitrary locations.

Canvas separates remote memory of each `cgroup` to isolate capacity and performance. The user creates a `cgroup` to set a size limit of remote memory for an application. Canvas allocates remote memory in a demand-driven manner—upon a pressure in local memory, Canvas allocates remote memory and registers it as a RDMA buffer. Canvas enables per-`cgroup` swap partitions by creating a swap partition interface and attaching it to each `cgroup`. For each `cgroup`, a separate swap-entry manager is used for allocating and freeing swap entries. Swap entry allocation can now be charged to the `cgroup`, which controls how much remote memory each application can use. Our adaptive swap entry allocation algorithm is discussed in §5.1.

Canvas explicitly enables a private swap cache for each `cgroup` (a default value of 32MB), whose size is charged to the *memory budget* specified in the `cgroup`. As a result, the size of an application's swap cache changes in response to its own memory usage, without affecting other applications.

For each demand swap-in, Canvas first checks the `mapcount` of the page, which indicates how many processes this page has been mapped to before. If the page belongs only to one process, it is placed in its private swap cache. Otherwise, it has to be placed in a global swap cache (discussed shortly). To release pages (*e.g.*, when the application's working set increases, pushing the boundary of the swap cache), Canvas scans the swap cache's page list, releasing a batch of pages to shrink the cache.

RDMA Bandwidth Isolation. For each `cgroup`, Canvas isolates RDMA bandwidth with a set of *virtual* RDMA queue pairs (VQPs) and a centralized packet scheduler. Users can set the swap-in/swap-out RDMA bandwidth of a `cgroup` with our extended interface. Our RDMA scheduler works in two dimensions. The *first dimension* schedules packets across applications, while the *second dimension* schedules on a per-application basis—each `cgroup` has its *sub-scheduler* that schedules packets that belong to the `cgroup` between demand swapping and prefetching.

VQPs are high-level interfaces, implemented with lock-free linked lists. Each `cgroup` pushes its requests to the head of its VQP, while the scheduler pops requests from their tails. At the low level, our scheduler maintains three physical queue pairs (PQP) per core, for *demand swap-in*, *prefetching*, and *swap-out*, respectively. The scheduler polls all VQPs and forwards packets to the corresponding PQPs, using a *two-dimensional* scheduling algorithm (see §5.3).

Handling of Shared Pages. Processes can share pages due to shared libraries or memory regions. These pages cannot go to any private swap cache. Canvas maintains a global swap partition and cache for shared pages. When a page is evicted and ummapped, Canvas checks its `mapcount` and adds it to the global swap cache if the page is shared between different processes. All pages in the global swap cache will be eventually swapped out to the global partition using the original lock-based allocation algorithm. Conversely, pages swapped in (and prefetched) from the global swap partition are all placed into the global swap cache. For typical cloud applications such as Spark, Cassandra and Neo4j, the number of shared pages is much smaller than process-private pages, using locks in a normal way would not incur a large overhead. We cannot charge applications' `cgroups` for pages in the global swap cache, because which process(es) share these pages is unknown before they get mapped into pro-

Problem Description	Performance Impact	Canvas's Solution
Unlimited use of swap and RDMA resources	Apps generating higher swap thruput use disproportionately more resources	Holistic isolation of swap system RDMA isolation and scheduling (§4, §5.3)
Lock conten. at swap entry alloc.	Reduced swap-out thruput	(1) Swap parti. isolation (§4); (2) adaptive entry alloc. (§5.1)
Single low-level prefetcher	Increased fault-handling latency	Two-tier adaptive prefetching (§5.2)
prefetching v.s. demand interfere	Increased fault-handling latency	Two-dimensional RDMA scheduling (§5.3)

Table 1: Summary of major issues and Canvas's solution.

cesses' address spaces. Canvas allows users to create a special `cgroup`, named `cgroup-shared`, to limit the size of the global swap cache/partition.

One limitation of our `cgroup`-based approach is that `cgroup` can only partition resources statically while applications' resource usage may change from time to time and static partitioning could lead to resource underutilization. However, the focus of this paper is to ensure isolation and future work could incorporate max-min fair allocation to improve resource utilization.

5 Isolation-Enabled Swap Optimizations

On top of the isolated swap system, we develop three optimizations, which dynamically adapt their strategies to each application's resource patterns and semantics.

5.1 Adaptive Swap Entry Allocation

As discussed in §3, swap entry allocation suffers from severe lock contention under frequent remote accesses—allocation is needed at every swap-out. To further motivate, we use a simple experiment by running Memcached alone on remote memory with different core numbers. As the number of cores increases, the average entry allocation time grows super-linearly—it grows from 10μs under 16 cores quickly to 130μs under 48 cores due to increased lock contention (see Figure 16). Creating a per-application swap partition mitigates the problem to a certain degree. However, applications like Spark run more than 90 threads; frequent swaps in these threads can still incur significant locking overhead.

To further reduce contention, we develop a novel swap entry allocator that adapts allocation strategies in response to each application's own memory access/usage. Our first idea is to enable a *one-to-one* mapping between pages and swap entries. At the first time a page is swapped out, we allocate a new swap entry using the original (lock-protected) algorithm. Once the entry is allocated, Canvas writes the entry ID into the page metadata (*i.e.*, `struct page`). This ID remains on the page throughout its life span. As a result, subsequent swap-outs of the page can write data directly into the entry corresponding to this ID. We pay the locking overhead *only once* for each page at its first swap-out.

This approach requires a swap entry to be reserved for each page. For example, if the local memory size is S and the remote memory allocation is 3S, with one-to-one mapping the remote memory allocation would be 4S (*i.e.*, each page residing in local memory also has a remote page, resulting in a 33% overhead). However, this overhead may not be necessary. For example, modern applications exhibit strong epochal behaviors. Under the original allocator, swap entries for pages accessed in one epoch can be reused for those in another epoch. Under this approach, however, all pages in all epochs must have their dedicated swap entries throughout the execution, which can lead to an order-of-magnitude increase in remote memory usage.

Our key insight is: we should trade off *space for time* if an application has much available swap space, but *time for space* when its space limit is about to be reached. As such, when the remote memory usage is about to reach the limit specified in `cgroup` (*i.e.*, 75% in our experiments), Canvas starts removing reservations to save space. The next question is which pages we should consider first as our candidates for reservation removal. Our idea is that we should first consider "hot pages" that always stay in local memory and are rarely swapped. This is because hot pages (*i.e.*, data on such pages are frequently accessed) are likely to stay in local memory for a long time; hence, locking overhead is less relevant for them. On the contrary, "cold" pages whose accesses are *spotty* are more likely to be swapped in/out and hence swap efficiency is critical. Here "hot" and "cold" pages are relatively defined as they are specific to execution stages—a cold page swapped out in a previous stage can be swapped in and become hot in a new stage.

To this end, we develop an *adaptive allocator*. Canvas starts an execution by reserving swap entries for *all* pages to minimize lock contention. Reservation removal begins when remote-memory pressure is detected. Canvas adaptively removes reservations for hot pages. We detect hot pages *for each application* by periodically scanning the application's *LRU active list*—pages recently accessed are close to the head of the active list. Each scan identifies a set of pages from the head of the list; a page is considered "hot" if it appears in a consecutive number of sets recently identified.

Removing the reservation for a hot page can be done by (1) removing the entry ID from the page metadata and (2) freeing its reserved swap entry in remote memory, adding the entry back to the free list. Once a hot page becomes cold and gets evicted, it does not have a reservation any more, and hence, it goes through the original (lock-protected) allocation algorithm to obtain an entry. In this case, the page

receives a new swap entry and remembers this new ID in its metadata.

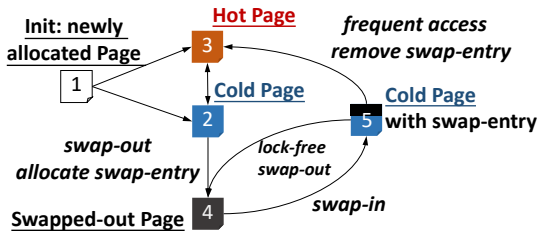


Figure 7: FSM describing our page management when remote-memory pressure is detected.

Figure 7 shows the page state machine, which describes the page handling logic. A cold page (to be evicted) can be in one of the two states: state 2 and state 5. A page comes to state 2 if it is (1) a brand new page that has never been swapped out or (2) previously a hot page but has not been accessed for long. Once it reaches state 2, the page does not have a reserved swap entry ID and hence, swapping out this page goes through the normal allocation path. In the case of swap-in (state 5), the swap entry ID is already remembered on the page. The next swap-out will directly use this entry and be lock-free. If the page becomes hot (from state 5 to 3), Canvas removes the entry ID and releases the entry reservation. The entry is then added back to the free list.

Performance Analysis. To understand the performance of the adaptive entry allocation algorithm, let us consider the following two scenarios. In the first scenario, the application performs uniformly random accesses. As a result, Canvas cannot clearly distinguish hot/cold pages, and thus randomly cancels their reservations. However, due to the random process, when a page is swapped out, it has a certain probability of still possessing a reserved swap entry (depending on the ratio of remaining reservations) and hence Canvas can still improve the allocation performance.

In the second scenario, the application follows a repetitive pattern of accessing a page a few times (making it hot) and then moving on to accessing another page; it will not come back to the page in a long while. Under our allocation algorithm, every page will be identified as a hot page, leading to the cancellation of its reservation. However, each page will be swapped out when it is cold enough; at each swap-out, the page has to go through the original allocation algorithm. This is the worst-case scenario, and even in this case, Canvas has the same (worst-case) performance as the original Linux allocator, which allocates an entry at each swap-out.

Some of the recent patches submitted to the Linux community also attempt to reduce lock contention for swap entry allocation. A detailed description of how Canvas differs from these patches can be found in Appendix B.

5.2 Two-tier Adaptive Prefetching

Problems with Current Prefetchers. Current prefetchers all focus on low-level (streaming or strided) access patterns. While such patterns exist widely in native array-based programs, applications written in high-level languages such as Python and Java are dominated by reference-based data structures—operations over such data structures involve large amounts of pointer chasing, making it hard for current prefetchers to identify clear patterns.

Furthermore, cloud applications such as Spark are heavily multi-threaded. Modern language runtimes, such as the JVM, run an additional set of auxiliary threads, *e.g.*, for GC or JIT compilation. How these user-level threads map to kernel threads is often implemented differently in different runtimes. Consequently, kernel prefetchers such as Leap [73] cannot distinguish patterns from different threads.

To develop an adaptive prefetcher, Canvas employs a two-tier design, illustrated in Figure 8. At the low (kernel) tier, Canvas uses an existing kernel prefetcher that prefetches data for each application into its own private swap cache (unless data comes from the global swap partition). A kernel prefetcher is extremely efficient and can already cover a range of (array-based) applications. For applications whose accesses are too complex for the kernel prefetcher to handle, we forward the addresses up to the application level, letting the application/runtime analyze semantic access patterns at the level of threads, references, arrays, *etc.*

Prefetching Logic. In Canvas, we adopt the sync/async separation design in Fastswap [8], which prevents head-of-line blocking. As stated earlier, we use three PQPs per core, one for swap-out, one for (sync) demand swap-in, and one for (async) prefetching. Canvas polls for completions of critical (demand) operations, while configuring *interrupt completions* for asynchronous prefetches.

Canvas determines whether to use an application-tier prefetcher based on *how successful kernel-tier prefetching is*. If the number of pages prefetched for an application is lower than a threshold at the most recent N (=3 in our evaluation) faults consecutively, Canvas starts forwarding the faulting addresses up to the application-tier prefetcher (discussed shortly) although the kernel-tier prefetcher is still used as the first-line prefetcher.

Canvas stops forwarding whenever the kernel-tier prefetcher becomes effective again. Our key insight is: the kernel-tier prefetcher is efficient without needing additional compute resources (as it uses the same core as the faulting thread), while the application-tier prefetcher needs extra compute resources to run. As such, we disable application-tier prefetchers as long as the kernel-tier prefetcher is effective. To pass a faulting address to the application, we modify the kernel's `userfaultfd` interface, allowing applications to handle faults at the user space. Our modification makes the kernel forward the faulting address only if the kernel's prefetcher continuously fails to prefetch pages.

Runtime Support for Application-tier Prefetching. A major challenge is how to develop application-tier prefetchers. On the one hand, application-tier prefetchers should conduct

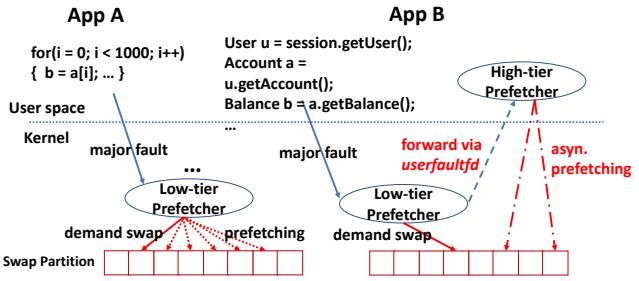


Figure 8: Canvas's two-tier prefetcher: App A is an array-based program while B is a modern web application that uses reference-based data structures. The low-tier prefetcher successfully prefetches pages for A, but not for B. Hence, Canvas forwards the addresses up to B's high-tier prefetcher.

prefetching based on *application semantics*, of which the kernel is unaware. On the other hand, application developers may not be familiar with a low-level activity like prefetching; understanding memory access patterns and developing prefetchers can be a daunting task for them.

Our insight is: applications that benefit from application-tier prefetching are mostly written in high-level languages and run on a managed runtime such as the JVM. Inspired by previous work on using language runtime to solve memory efficiency problems for data analytics applications [81, 78, 82, 80, 72], Canvas currently supports application-tier prefetching for the JVM as a platform. However its support could be easily extended to other managed runtimes for high-level languages like Go and C#. Leveraging language runtime solves both problems discussed above—it has access to semantic information such as how objects are connected and the number of application threads; furthermore, the burden of developing an application-tier prefetcher is shifted from application developers to runtime developers. Thus, it is not necessary to supply a custom application-tier prefetcher per application, but define it once for each language runtime.

In this work, we develop an application-tier prefetcher in Oracle's OpenJDK as a proof-of-concept. It works for all (Java, Scala, Python, *etc.*) programs that run on the JVM. Our JVM-based prefetcher considers two *semantic patterns*: (1) *reference-based* (*i.e.*, accessing an object brings in pages containing objects referenced by this object) and (2) *thread-based* (*i.e.*, accesses from different application threads are separately analyzed to find patterns).

For (1), we modify the JVM to add support that can quickly find, from a faulting address, *the object* in which the address falls. We use write barrier, a piece of code instrumented by the JVM at each object field write, as well as the garbage collector to record references between pages. For example, for each write of the form `a.f=b`, if the objects referenced by a and b are on different page groups, we record an edge on a *summary graph* where each node represents a consecutive group of pages and each edge represents references between groups. During prefetching, we traverse the graph from the node that represents the accessed page and prefetch pages that can be reached within 3 hops. The traversal does not follow cycles and its overhead is negligible. This approach is suitable for applications that store a large amount of data in memory, such as Spark and Cassandra.

For (2), we leverage the JVM's user-kernel thread map. For each faulting address, Canvas additionally forwards the thread information (*i.e.*, pid) to the JVM, which consults the map to filter out non-application (*e.g.*, GC, compilation, *etc.*) threads and segregate addresses based on Java threads (as opposed to kernel threads). Segregated addresses allow us to analyze (sequential/strided) patterns on a per-thread basis (using Leap's majority-vote algorithm [73]). Once patterns are found, the prefetcher sends the prefetching requests to the kernel via `async_prefetch`.

For native programs that directly use kernel threads (*e.g.*, `pthread`), the thread information is straightforward and immediately visible to Canvas. We can easily segregate addresses accessed from different threads and analyze patterns based upon addresses from each individual thread.

Policy. To improve effectiveness, the JVM uses a search tree to record information about large arrays. Upon the allocation of an array whose size exceeds a threshold (*i.e.*, 1MB in our experiments), the JVM records its starting address and size into the tree. The JVM runs a daemon prefetching thread. Once receiving a sequence of faulting addresses, we determine which semantic pattern to use based on *how many application threads are running* and *whether the faulting addresses fall into a large array*. If there are many threads and the faulting addresses fall into arrays, the JVM uses (2) to find per-thread patterns. If either condition does not hold, the JVM uses (1) to prefetch based on references. For native applications, we only enable (2), as we observed that our native programs do not use many deep data structures.

5.3 Two-Dimensional RDMA Scheduling

To provide predictable performance for applications sharing RDMA resources, our RDMA scheduling algorithm should provide four properties: (1) weighted fair bandwidth sharing [18, 30] across applications; (2) high overall utilization; (3) treating demand and prefetching requests with different priorities; and (4) timely handling of prefetching requests.

Canvas performs two-dimensional scheduling by extending existing techniques. Canvas uses max-min fair scheduling to assign bandwidth across applications, and priority-based scheduling with *timeliness* to schedule prefetching and demand requests within each application. Although these scheduling techniques are not new themselves, Canvas combines them in a unique way to solve the interference problem. Canvas maintains three PQPs on each core, respectively, for swap-outs, demand swap-ins, and prefetching swap-ins. Swap-outs are only subject to fair scheduling while swap-ins are subject to both fair and priority-based scheduling.

Vertical: Fair Scheduling. Under max-min fairness, each application receives a fair share of bandwidth. If there is extra bandwidth, we give it to the applications in the reverse order of their bandwidth demand until bandwidth is saturated. The high overall utilization of bandwidth is achieved by redistributing unconsumed bandwidth proportionally to the weights of unsatisfied applications. Canvas implements weighted fair queuing with virtual clock [84, 30, 110].

Horizontal: Priority Scheduling with Timeliness. Within each `cgroup`, Canvas schedules demand requests with a higher priority than prefetching requests. However, this could lead to long latency for prefetching requests. To bound the latency of prefetching, our scheduler employs a history-based heuristic algorithm to identify and drop outdated prefetching requests. In particular, Canvas maintains the *timeliness distribution* of prefetched pages per `cgroup`. Timeliness is a metric that measures the time between a page being prefetched and accessed. We attach a timestamp to each request when pushing it into a VQP. The scheduler maintains packets statistics on-the-fly to estimate the round-trip latency and arrival time of each prefetching request. Requests are dropped if the estimated arrival time exceeds the estimated timeliness threshold.

Special care must be taken to drop prefetching requests. Before issuing a prefetching request, the kernel creates a page in the swap cache and sets up its corresponding PTE. The page is left in a *locked* state until its data comes back. However, a thread that accesses an address falling into the page may find this locked page in the swap cache and block on it. Dropping prefetching requests may cause the thread to hang. To solve the problem, we detect threads that block on prefetching requests for too long and generate new *demand requests* for them.

We rely on a per-entry timestamp to efficiently detect threads that block on prefetching requests. In Canvas, we attach a timestamp field to the swap entry metadata. Canvas's scheduler records the timestamp every time it enqueues a prefetching request into VQP. If another thread faults on the same page later, it will retrieve the same swap entry from the PTE. If the swap entry contains a timestamp, the faulting thread knows that a prefetching request has already been issued. Next, the faulting thread calculates the time elapsed since the timestamp, and compares it with a timeout threshold (maintained by the RDMA scheduler based on page-fetching latencies). If it exceeds the timeout threshold, the faulting thread drops the prefetching request. The drop operation is elaborated below:

Before issuing each (demand or prefetching) request, the kernel first allocates a physical page in the swap cache and locks the page until the request returns. Upon the return of the data, the data is written into the page; the page is unlocked and mapped into the page table. In order to safely drop a request, we add another field *valid* in the swap entry metadata, indicating whether the prefetching request on the go is valid. Once a faulting thread identifies a delayed prefetching request (by using the timestamp as discussed above), it sets the *valid* field in the swap entry to `false` and then creates a new physical page in the swap cache. The thread goes ahead and issues another (demand) I/O request based on this new page. When the delayed prefetching request returns, it checks the *valid* field and discards itself once it sees the false value. The field is then set back to `true`.

When a demand request is issued, Canvas clears the timestamp field in its corresponding swap entry. If a thread faults on the same page, it will block on the request instead of issuing a new one due to the empty timestamp (indicating that the request on the go is a demand one).

6 Evaluation

It took us 17 months to implement Canvas in Linux 5.5. The application-tier prefetcher was implemented in OpenJDK 12.

Application	Workload	Dataset	Size / ($\|E\|$, $\|V\|$)
Managed			
Cassandra	5M read, 5M insert	YCSB[26]	10M records
Neo4j	PageRank	Baidu[5]	(17M, 2M)
Spark	PageRank (**SPR**)	Wikipedia[5]	(57M, 1.5M)
	KMeans (**SKM**)	Wikipedia[5]	188M points
	Logistic Regression (**SLR**)	Wikipedia[5]	188M points
	Skewed Groupby (**SSG**)	synthetic	256K records
	Triangle Counting (**GTC**)	synthetic	(1.5M, 384K)
MLlib	Bayes Classifiers (**MBC**)	KDD [3]	1.5M instances
GraphX	Connected Components (**GCC**)	Wikipedia[5]	(188M, 9M)
	PageRank (**GPR**)	Wikipedia[5]	(188M, 9M)
	Single Src. Shortest Path (**GSP**)	synthetic	2M vertices
Native			
XGBoost	Binary Classification	HIGGS[12]	22M instances
Snappy	Compression	enwik9 [1]	16GB
Memcached	45M gets, 5M sets	YCSB[26]	10M records

Table 2: Programs and their workloads.

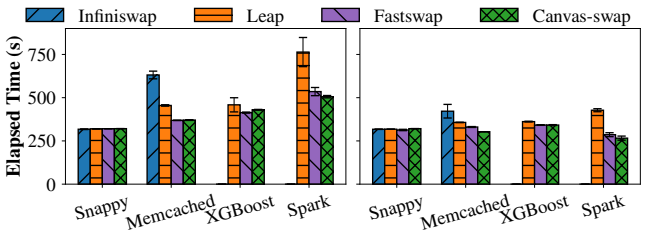
Setup. We included a variety of cloud applications in our experiments, including managed (Java) applications such as Spark [109], Cassandra [10] (a NoSQL database), Neo4j [79] (a graph database), as well as three native applications: XGBoost [23], Snappy [38], and Memcached [4]. Spark, Cassandra, Neo4j, Memcached, and XGBoost are multi-threaded while Snappy is single-threaded. The Spark applications span popular libraries such as GraphX and MLlib.

We co-ran different combinations of programs. The same application in different combinations receives the same amount of local (CPU and memory) resources. To simplify performance analysis, we let each combination of applications co-run contain one managed (Spark, Cassandra, or Neo4j) application and the three native programs, which consume less resources. These experiments were conducted on two machines, one used to execute applications and a second to provide remote memory. The configurations of these machines was reported earlier in §3. We carefully configured Linux with the following configuration to achieve the best performance for Linux: (1) SSD-like swap model, (2) per-VMA prefetching policy, and (3) cluster-based swap en-

try allocation. We disabled hyper-threads, CPU C-states, dynamic CPU frequency scaling, transparent huge pages, and the kernel's mitigation for speculation attacks.

For each combination, we limited the amounts of CPU resources for the managed application, XGBoost, Memcached, and Snappy to be 24, 16, 4, and 1 core(s). For local memory, we used two ratios: 50% and 25%, meaning each application has 50/25% of its working set locally. When using Canvas, we additionally limited the sizes of swap partitions in such a way that for each application the total size of its swap partition and assigned local memory is slightly larger than its working set. In doing so, each application has just enough (local and remote) memory to run and reservation cancellation (§5.1) is triggered in all executions.

The swap cache size for each application starts at 32MB and changes dynamically. The global swap cache size (configured by `cgroup-share`) was also set to 32MB. Canvas uses max-min fair scheduling to assign bandwidth across applications, and their initial weights are proportional to their swap partition assignments. We ran each application 10 times. Their average execution times (with error bars) are reported in all experiments throughput this section.



(a) 25% local memory. (b) 50% local memory.

Figure 9: Performance of different swap systems.

6.1 Basic Swap Systems

We used Fastswap [8] as our underlying swap system, with a small amount of code changes to port Fastswap (originally built against Linux 4.11) to Linux 5.5. We first compared the performance of each individual application running on basic swap systems including Infiniswap [39], Infiniswap with Leap [73], the original Fastswap [8], and Canvas's ported Fastswap (without isolation and optimizations). We could not run LegoOS [91] as it does not support network-related system calls, which are required for applications such as Spark. LegoOS implements swaps with RPCs as opposed to paging, but our idea (*i.e.*, isolation and adaptive swapping) is applicable to this approach as well.

We ran Infiniswap and Leap on Linux 4.4, and Fastswap on Linux 4.11. The results are reported in Figure 9. Infiniswap hung on XGBoost and Spark, and its corresponding bars are thus not reported in Figure 9. Since Canvas-swap was built off Fastswap, they have similar performance.

6.2 Overall Performance

Next, we demonstrate the overall performance when applications co-run together under Canvas. Each experiment ran the same set of three native programs with one managed application: Spark-LR, Spark-KM, Cassandra, or Neo4j. The results for the 25% and 50% local memory configurations are reported in Figure 10(a) and (b), respectively.

The four bars in each group represent an application's performance when running alone on Linux 5.5, co-running with other applications on Linux 5.5, co-running on the original Fastswap, and co-running on Canvas (with all optimizations enabled). Across all experiments, Canvas improves applications' co-run performance by up to **6.2**× (average **3.5**×) and up to **3.8**× (average **1.9**×) under the two memory configurations. Canvas enables Spark and Neo4j to even outperform their individual runs due to the optimizations that could also improve single-application performance.

6.3 Isolation Reduces Degradation and Variation

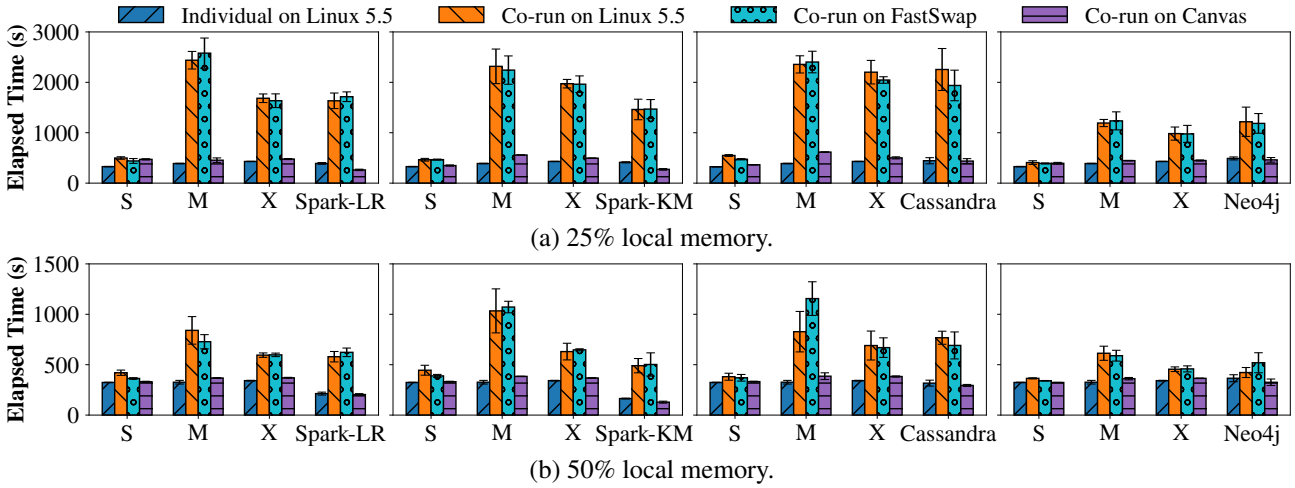
This experiment measures the effectiveness of isolation alone. We used a variant of Canvas with the isolated swap system and RDMA bandwidth (*i.e.*, vertical scheduling between applications) but without our swap-entry optimization, two-tier prefetcher, and horizontal RDMA scheduling.

Degradation Reduction. We ran the same set of experiments under 25% local memory. As shown in Figure 11, isolation reduces the running time by up to 5.2×, with an average of 2.5×. Isolation is particularly useful for applications that do not have many threads but need to frequently access remote memory, such as Memcached, which has 4 threads and cannot compete for resources with managed applications such as Spark and Cassandra, which have more than 90 (application and runtime) threads. As such, its performance is improved by **3.3**× with dedicated swap resources. Isolation improves the average RDMA utilization by 2.8× from 692MB/s to 1908MB/s, making the peak bandwidth reach 4494MB/s.

Table 3: Performance variations of three native applications when co-running with each of the 11 managed applications under 25% local memory (Canvas / Linux 5.5 / Fastswap).

Program	Mean			Min			Max			σ		
Snappy	**1.07**	1.28	1.23	**1.03**	1.10	1.08	**1.23**	1.69	1.46	**0.07**	0.20	0.14
Memcached	**1.45**	3.24	3.76	**1.30**	1.48	2.05	**1.91**	6.05	8.17	**0.20**	1.82	2.14
XGBoost	**1.05**	3.17	2.81	**1.01**	1.38	1.91	**1.13**	6.13	4.76	**0.04**	1.59	1.11
Overall	**1.21**	2.56	2.60	**1.01**	1.10	1.08	**1.91**	6.13	8.17	**0.23**	1.64	1.72

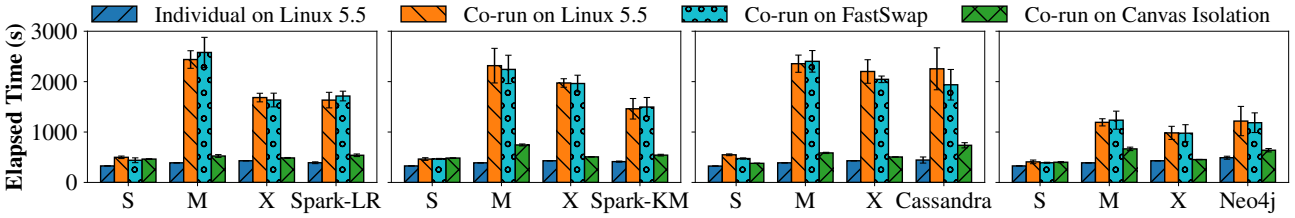
Variation Reduction. One significant impact of interference is performance variation—the same application has drastically different performance when co-running with different applications (as shown in Figure 2). To demonstrate our benefits, we co-ran the three native applications with each of the eleven managed applications listed in Table 2, which cover a wide spectrum of computation and memory access behaviors. Table 3 reports various statistics of their perfor-



(a) 25% local memory.

(b) 50% local memory.

Figure 10: Performance of each program under 25% and 50% local memory when the three native programs, Snappy (S), Memcached (M), and XGBoost (X), co-run with a managed application. Canvas ran with all optimizations enabled.



(a) Co-run with Spark-LR. (b) Co-run with Spark-KM. (c) Co-run with Cassandra. (d) Co-run with Neo4j.

Figure 11: Performance of native applications co-run with different managed applications under 25% local memory; for Canvas, only isolation was enabled (*i.e.*, without adaptive optimizations).

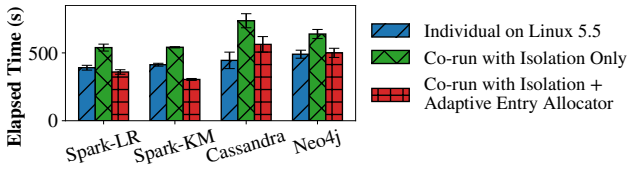


Figure 12: Benefit of adaptive swap entry allocation. Compared are the times of the application running individually on Linux 5.5, co-running on Canvas with adaptive entry allocation disabled, and enabled.

mance including the mean, minimum, maximum, and standard deviation of their slowdowns (compared to their individual runs). Clearly, the performance of the three programs is much more stable (indicated by a small σ) under Canvas than Linux—variations are reduced by 7× overall.

6.4 Effectiveness of Adaptive Optimizations

This subsection evaluates the benefit of each swap optimization *on top of the isolated swap system* by turning it on/off.

6.4.1 Adaptive Swap Entry Allocator

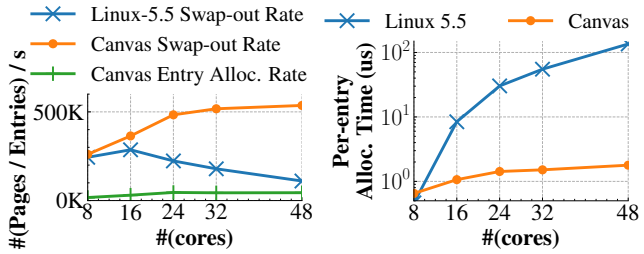
Isolation already reduces lock contention at swap entry allocation because each process has its own swap entry manager. However, for multi-threaded applications such as Spark and Cassandra, their processing threads still have to go through the locking process. In this subsection, we focus on managed applications due to their extensive use of threads. Figure 12 shows the performance of Spark LR, Spark KM, Cassandra, and Neo4j when they each co-run with the other three native programs. On average, our adaptive allocation enables an *additional* boost of 1.50× for Spark LR, 1.77× for Spark KM, 1.31× for Cassandra, and 1.28× for Neo4j.

Table 4: Swap-out throughput w/ and w/o adaptive swap-entry allocation when native programs co-run with Spark.

Thruput (KPages/s)	Linux 5.5	Canvas w/o adap. alloc.	Canvas w/
Avg. Spark apps	98	164	**295**
Avg. all apps	185	309	**468**

Table 4 reports the swap-out throughput when the native applications co-run with Spark. As shown, isolation improves the throughput by 1.67× while adaptive allocation provides an additional boost of 1.51×. This benefit is obtained after applying all optimizations in Linux 5.5.

Effectiveness of Entry Reservation. We compared our adaptive allocation algorithm with the original allocator in Linux 5.5 by running Memcached with varying (8 – 48) cores under 25% local memory. As shown in Figure 13(a), for Canvas,



(a) Swap-out and entry alloc rates. (b) Per-entry alloc time.

Figure 13: Entry allocation comparison between the allocation algorithm in Canvas and Linux 5.5 for Memcached under 25% local memory. The Y-axis in (b) is log-scaled.

(1) the swap-out rate increases with the core number (showing good scalability) and (2) the swap entry allocation rate remains low. This is due to Canvas's entry reservation algorithm that effectively reuses a significant number of swap entries for page swap-outs. On the contrary, in Linux 5.5, the swap-out rate (which is the same as its entry allocation rate) decreases when more cores are used. This is because each entry allocation takes significantly longer, reducing the swap-out throughput. A comparison of per-entry allocation time can be seen in Figure 13(b). We additionally compared the allocation algorithm between Canvas, Linux 5.5, and Linux 5.14; these results are reported in Appendix B.

6.4.2 Prefetching Effectiveness

Our baseline is the kernel's default prefetcher on the isolated swap system with adaptive swap allocator *enabled*. Since application-tier prefetching is designed primarily for high-level languages, here we focus on managed programs.

Time. We compare the running time for three Spark applications LR, KM, TC, and Neo4j, between the kernel's prefetcher over Canvas's isolated swap system and Canvas's two-tier prefetcher, when each managed application co-runs with the three native applications under the 25% local memory configuration. Application-tier prefetching brings **33%**, **17%**, **19%**, and **8%** additional performance benefits on top of the kernel prefetching with the isolated swap system. All the four managed applications benefit from the thread-level pattern analysis while the managed applications have seen 5-9% contributions from using the reference-based pattern. The thread-level pattern analysis we added for native programs brings a 5% and 11% improvement for Memcached and XGBoost.

We have also run Leap [73], a prefetcher that aggressively prefetches a number of contiguous pages if it cannot find any pattern. This approach may work for native applications because these applications access arrays; hence, the contiguous pages aggressively prefetched are likely to be useful for array accesses. However, it works poorly for high-level language applications such as Spark and Neo4j, which use deep data structures and run graph-traversal GC tasks (which exhibit neither sequential nor strided patterns). Aggressively prefetching useless pages wastes the RDMA bandwidth and the swap cache. Leap slows down our managed applications by 1.4×, compared to the kernel's default prefetcher.

Table 5: Prefetching contribution and accuracy when different Spark and Neo4j co-run with native applications.

Contribution	Spark-LR	Spark-KM	Spark-TC	Neo4j
Leap	23.4%	25.8%	42.2%	**67.0%**
Kernel	63.3%	68.0%	65.9%	41.1%
Canvas Two-tier	**79.2%**	**79.3%**	**75.3%**	45.0%
Accuracy	**Spark-LR**	**Spark-KM**	**Spark-TC**	**Neo4j**
Leap	16.8%	17.2%	35.9%	6.1%
Kernel	95.6%	96.4%	93.9%	80.4%
Canvas Two-tier	**94.3%**	**94.8%**	**94.9%**	**87.1%**

Prefetching Contribution and Accuracy. Table 5 compares prefetching *contribution* and *accuracy* for the four managed applications when each of them co-runs with the same three native applications. Contribution is defined as a ratio between the number of page faults hitting on the swap cache and the total number of page faults (including both cache hits and demand swap-ins). Accuracy is defined as a ratio between the number of page faults hitting on the swap cache and the total number of prefetches. Clearly, contribution has a strong correlation with performance while accuracy measures the pattern recognition ability of a prefetcher. For example, for a conservative prefetcher that prefetches pages only if a pattern can be clearly identified, it can have a high accuracy (*i.e.*, prefetched pages are all useful) but a low contribution (*i.e.*, the number of prefetches is small).

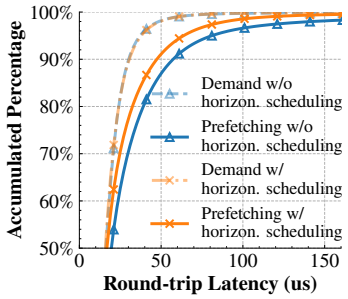
Here we report prefetching contribution and accuracy for three prefetchers: Leap (on our isolated swap system), the kernel prefetcher (also on our isolated swap system), and Canvas's two-tier prefetcher. Among the three prefetchers, for all but Neo4j, Leap has the lowest accuracy and contribution because it is an aggressive prefetcher. Leap keeps prefetching pages even when it cannot detect any patterns, which greatly reduces the prefetching accuracy. Second, due to the limited swap cache, the useless pages prefetched can cause previously prefetched pages to be released before they are accessed, hurting contribution. The kernel prefetcher and Canvas have comparable accuracy because the kernel prefetcher is much more conservative than Leap. It stops prefetching when no clear pattern can be observed. However, Linux has lower contribution than our two-tier prefetcher since Canvas prefetches more useful pages using semantics.

6.4.3 RDMA Scheduling

We evaluate our two-dimensional RDMA scheduling. For the vertical dimension, we use the weighted min-max ratio (WMMR) $\frac{\min(x_i/w_i)}{\max(x_i/w_i)}$ [96] as our bandwidth fairness metric (the closer to 1, the better), where x_i is the bandwidth consumption of the application i, and w_i is its weight. We set the

weight proportionally to the average bandwidth of each application when running individually. Our vertical scheduling achieves an overall of **0.88** WMMR.

The horizontal dimension (*i.e.*, priority scheduling with timeliness) is our focus here because interference between prefetching and demand swapping is a unique challenge we overcome in this work. We ran GraphX Connected Components (GraphX-CC) with the three native applications. Figure 14 compares the latency of sync vs. async swap-in requests with and without the horizontal scheduling of RDMA.



w/o horizontal scheduling
Contribution: 52.1%
Accuracy: 46.9%
w/ horizontal scheduling
Contribution: **62.8%**
Accuracy: **52.4%**

(a) Latency CDF. (b) Prefetching effectiveness.

Figure 14: Horizontal scheduling effectiveness for GraphX-CC: (a) prefetching latency reduced, and (b) prefetching contribution and accuracy improved.

As shown, our scheduler does *not* incur overhead for the synchronous, demand requests but reduces the (90th percentile) latency of the asynchronous prefetching requests by ∼5%. Note that these results were obtained with Canvas's two-tier prefetcher enabled, which already generates precise prefetching requests. With the Leap prefetcher, the (90th percentile) latency reduction can be as high as **9**×. To understand how the latency reduction improves prefetching effectiveness, we have also compared the prefetching contribution and accuracy with and without the horizontal scheduling, as shown in Figure 14(b). Due to the high timeliness requirement of prefetching requests, even 5% latency reduction can lead to noticeable improvements in prefetching—*e.g.*, the contribution/accuracy of GraphX-CC increases by **10.7%** and **5.5%** on top of the two-tier prefetcher—which ultimately translate to a **7-12%** overall improvement.

7 Related Work

Remote Memory. The past few years have seen a proliferation of remote-memory systems that built on the kernel's swap mechanisms (including recent works such as LegoOS [91], Infiniswap [39], Fastswap [8], and Semeru [104] as well as earlier attempts [32, 6, 31, 34, 28, 45, 61, 105]). Remote memory is part of a general trend of resource disaggregation in datacenters [43, 21, 36, 14, 11, 66, 65, 58, 7, 9, 83, 95], which holds the promise of improving resource utilization and simplifying new hardware adoption. Under disaggregated memory, application data are stored on memory servers, making swap interference a more serious problem.

Resource Isolation. Interference exists in a wide variety of settings [29, 69, 111] and resource isolation is crucial for delivering reliable performance for user workloads. There is a large body of work on isolation of various kinds of resources including compute time [64, 16, 25], processor caches [35, 57, 106], memory bandwidth [67, 68, 71, 50, 107], I/O bandwidth [40, 96, 70, 74, 97, 103, 108], network bandwidth [13, 41, 37, 94, 87, 77, 53], congestion control [27, 44], as well as CPU involved in network processing [59]. Techniques such as IX [17] and MTCP [52] isolate data-plane and application processing at the core granularity.

Prefetching. Prefetching has been extensively studied, in the design of hardware cache [101, 42, 114, 100, 76], compilers [98, 63, 89, 86, 60, 33], as well as operating systems [102, 73]. Detecting spatial patterns [75] is a common way to prefetch data. For example, various hardware techniques [93, 54, 51] have been developed to identify patterns (*i.e.*, sequential or stride) in addresses accessed. Leap [73] is a kernel prefetcher designed specifically for applications using remote memory. Swap interference can reduce the effectiveness of any existing prefetchers, let alone that none of them consider complex (semantic) patterns. Early work such as [85, 20] proposes application-level prefetching for efficient file operations on slow disks. Our prefetcher is, however, designed for a new setting where applications trigger page faults frequently and read pages from fast remote memory, with much tighter latency budgets.

RDMA Optimizations. There is a body of recent work on RDMA scheduling [88, 92] and scalability improvement [99, 24, 56, 55, 113]. These techniques focus more on scalability when RDMA NICs are shared among multiple clients.

8 Conclusion

We observed swap resources must be isolated when multiple applications use remote memory simultaneously. As such, Canvas isolates swap cache, swap partition, and RDMA bandwidth to prevent applications from invading each other's resources. Now that resource accounting is done separately for applications, Canvas offers three optimizations that adapt kernel operations such as swap-entry allocation, prefetching, and RDMA scheduling to each application's resource usage, providing additional performance boosts.

Acknowledgement

We thank the anonymous reviewers for their valuable and thorough comments. We are grateful to our shepherd Danyang Zhuo 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-2152313, CNS-2151630, and CNS-2140552, CNS-2153449, ONR grant N00014-18-1-2037, a Sloan Research Fellowship, and research grants from Cisco, Intel CAPA, and Samsung.

References

[1] Large Text Compression Benchmark.

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

[3] Libsvm data: Classification. https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets, 2012.

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

[5] Konect networks data. http://konect.cc/networks/, 2021.

[6] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, S. Novakovic, A. Ramanathan, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote regions: A simple abstraction for remote memory. In *USENIX ATC*, pages 775–787, 2018.

[7] M. K. Aguilera, K. Keeton, S. Novakovic, and S. Singhal. Designing far memory data structures: Think outside the box. In *HotOS*, pages 120–126, 2019.

[8] 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.

[9] S. Angel, M. Nanavati, and S. Sen. Disaggregation and the application. In *HotCloud*, 2020.

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

[11] K. Asanović, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[12] 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.

[13] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *SIGCOMM*, pages 242–253, 2011.

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

[15] L. A. Barroso, U. Hölzle, and P. Ranganathan. *The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition*. Synthesis Lectures on Computer Architecture, 2018.

[16] D. B. Bartolini, F. Sironi, D. Sciuto, and M. D. Santambrogio. Automated fine-grained cpu provisioning for virtual machines. *ACM Trans. Archit. Code Optim.*, 11(3), July 2014.

[17] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *OSDI*, pages 49–65, 2014.

[18] D. Bertsekas and R. Gallager. *Data Networks (2nd Ed.)*. Prentice-Hall, Inc., USA, 1992.

[19] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli. Rethinking software runtimes for disaggregated memory. In *ASPLOS*, pages 79–92, 2021.

[20] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Trans. Comput. Syst.*, 14(4):311–343, Nov. 1996.

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

[22] T. Chen and C. Guestrin. XGBoost: A scalable tree boosting system. In *KDD*, pages 785–794, 2016.

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

[24] Y. Chen, Y. Lu, and J. Shu. Scalable RDMA RPC on reliable connection with efficient resource sharing. In *EuroSys*, 2019.

[25] L. Cherkasova, D. Gupta, and A. Vahdat. Comparison of the three cpu schedulers in xen. *SIGMETRICS Perform. Eval. Rev.*, 35(2):42–51, 2007.

[26] 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, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[27] B. Cronkite-Ratcliff, A. Bergman, S. Vargaftik, M. Ravi, N. McKeown, I. Abraham, and I. Keslassy. Virtualized congestion control. In *SIGCOMM*, pages 230–243, 2016.

[28] 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 *OSDI*, 1994.

[29] C. Delimitrou and C. Kozyrakis. Bolt: I know what you did last summer... in the cloud. In *ASPLOS*, pages 599–613, 2017.

[30] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *SIGCOMM Comput. Commun. Rev.*, 19(4):1–12, Aug. 1989.

[31] M. J. Feeley, W. E. Morgan, E. P. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *SOSP*, pages 201–212, 1995.

[32] E. Felten and J. Zahorjan. Issues in the implementation of a remote memory paging system. In *University of Washington CSE TR CSE TR*, 1991.

[33] M. Ferdman, C. Kaynak, and B. Falsafi. Proactive instruction fetch. In *MICRO*, pages 152–162, 2011.

[34] M. D. Flouris and E. P. Markatos. The network ramdisk: Using remote memory on heterogeneous nows. *Cluster Computing*, 2(4), Dec 1999.

[35] L. Funaro, O. A. Ben-Yehuda, and A. Schuster. Ginseng: Market-driven LLC allocation. In *USENIX ATC*, pages 295–308, 2016.

[36] 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.

[37] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, pages 323–336, 2011.

[38] Google. Google's fast compressor/decompressor. https://github.com/google/snappy, 2020.

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

[40] A. Gulati, A. Merchant, and P. J. Varman. MClock: Handling throughput variability for hypervisor IO scheduling. In *OSDI*, pages 437–450, 2010.

[41] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. SecondNet: A data center network virtualization architecture with bandwidth guarantees. In *Co-NEXT*, 2010.

[42] Y. Guo. *Compiler-Assisted Hardware-Based Data Prefetching for next Generation Processors*. PhD thesis, 2007.

[43] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker. Network support for resource disaggregation in next-generation datacenters. In *HotNets*, pages 10:1–10:7, 2013.

[44] K. He, E. Rozner, K. Agarwal, Y. J. Gu, W. Felter, J. Carter, and A. Akella. AC/DC TCP: Virtual congestion control enforcement for datacenter networks. In *SIGCOMM*, pages 244–257, 2016.

[45] L. Iftode, K. Li, and K. Petersen. Memory servers for multicomputers. In *Digest of Papers. Compcon Spring*, pages 538–547, Feb 1993.

[46] Intel. Batch allocation for swap entries. https://github.com/torvalds/linux/commit/ed43af10975eef7e, 2020.

[47] Intel. Memcontrol: Charge swap-in pages to `cgroup`. https://github.com/torvalds/linux/commit/4c6355b25e8bb83c, 2020.

[48] Intel. Per-core cluster allocation. https://github.com/torvalds/linux/commit/490705888107c3ed, 2020.

[49] C. Iorgulescu, R. Azimi, Y. Kwon, S. Elnikety, M. Syamala, V. Narasayya, H. Herodotou, P. Tomita, A. Chen, J. Zhang, and J. Wang. PerfIso: Performance isolation for commercial Latency-Sensitive services. In *USENIX ATC*, pages 519–532, 2018.

[50] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. QoS policies and architecture for cache/memory in CMP platforms. In *SIGMETRICS*, pages 25–36, 2007.

[51] A. Jain and C. Lin. Linearizing irregular memory accesses for improved correlated prefetching. In *MICRO*, pages 247–259, 2013.

[52] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. MTCP: A highly scalable user-level TCP stack for multicore systems. In *NSDI*, pages 489–502, 2014.

[53] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, and C. Kim. EyeQ: Practical network performance isolation for the multi-tenant cloud. In *HotCloud*, 2012.

[54] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *ISCA*, pages 252–263, 1997.

[55] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high performance RDMA systems. In *USENIX ATC*, pages 437–450, 2016.

[56] 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.

[57] H. Kasture and D. Sanchez. Ubik: Efficient cache sharing with strict qos for latency-critical workloads. In *ASPLOS*, pages 729–742, 2014.

[58] K. Keeton. The Machine: An architecture for memory-centric computing. In *ROSS*, 2015.

[59] J. Khalid, E. Rozner, W. Felter, C. Xu, K. Rajamani, A. Ferreira, and A. Akella. Iron: Isolating network-based cpu in container environments. In *NSDI*, pages 313–328, 2018.

[60] A. Kolli, A. Saidi, and T. F. Wenisch. RDIP: Return-address-stack directed instruction prefetching. In *MICRO*, pages 260–271, 2013.

[61] S. Koussih, A. Acharya, and S. Setia. Dodo: a user-level system for exploiting idle memory in workstation clusters. In *HPDC*, pages 301–308, Aug 1999.

[62] 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.

[63] C. Lattner and V. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *PLDI*, pages 129–142, 2005.

[64] T. Li, D. Baumberger, and S. Hahn. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In *PPoPP*, pages 65–74, 2009.

[65] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *ISCA*, pages 267–278, 2009.

[66] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level implications of disaggregated memory. In *HPCA*, pages 1–12, 2012.

[67] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *PACT*, pages 367–376, 2012.

[68] L. Liu, Y. Li, Z. Cui, Y. Bao, M. Chen, and C. Wu. Going vertical in memory management: Handling multiplicity by multi-policy. In *ISCA*, pages 169–180, 2014.

[69] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Improving resource efficiency at scale with heracles. *ACM Trans. Comput. Syst.*, 34(2), 2016.

[70] L. Lu, Y. Zhang, T. Do, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Physical disentanglement in a container-based file system. In *OSDI*, pages 81–96, 2014.

[71] J. Ma, X. Sui, N. Sun, Y. Li, Z. Yu, B. Huang, T. Xu, Z. Yao, Y. Chen, H. Wang, L. Zhang, and Y. Bao. Supporting differentiated services in computers via programmable architecture for resourcing-on-demand (PARD). In *ASPLOS*, pages 131–143, 2015.

[72] M. Maas, K. Asanović, T. Harris, and J. Kubiatowicz. Taurus: A holistic language runtime system for coordinating distributed managed-language applications. In *ASPLOS*, pages 457–471, 2016.

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

[74] J. C. McCullough, J. Dunagan, A. Wolman, and A. C. Snoeren. Stout: An adaptive interface to scalable cloud storage. In *USENIX ATC*, 2010.

[75] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Trans. Comput. Syst.*, 2(3):181–197, 1984.

[76] S. Mittal. A survey of recent prefetching techniques for processor caches. *ACM Comput. Surv.*, 49(2), 2016.

[77] Y. Mundada, A. Ramachandran, and N. Feamster. Silverline: Data and network isolation for cloud services. In *HotCloud*, 2011.

[78] C. Navasca, C. Cai, K. Nguyen, B. Demsky, S. Lu, M. Kim, and G. H. Xu. Gerenuk: Thin computation over big native data using speculative program transformation. In *SOSP*, pages 538–553, 2019.

[79] Neo4j. Neo4j graph data platform. https://neo4j.com, 2021.

[80] K. Nguyen, L. Fang, C. Navasca, G. Xu, B. Demsky, and S. Lu. Skyway: Connecting managed heaps in distributed big data systems. In *ASPLOS*, pages 56–69, 2018.

[81] 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.

[82] K. Nguyen, K. Wang, Y. Bu, L. Fang, J. Hu, and G. Xu. Facade: A compiler and runtime for (almost) object-bounded big data applications. In *ASPLOS*, pages 675–690, 2015.

[83] 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.

[84] A. Parekh and R. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, 1993.

[85] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *SOSP*, pages 79–95, 1995.

[86] L. Peled, S. Mannor, U. Weiser, and Y. Etsion. Semantic locality and context-based prefetching using reinforcement learning. In *ISCA*, pages 285–297, 2015.

[87] L. Popa, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. Faircloud: Sharing the network in cloud computing. In *SIGCOMM*, pages 187–198, 2012.

[88] H. Qiu, X. Wang, T. Jin, Z. Qian, B. Ye, B. Tang, W. Li, and S. Lu. Toward effective and fair RDMA resource sharing. In *APNet*, pages 8–14, 2018.

[89] R. M. Rabbah, H. Sandanagobalane, M. Ekpanyapong, and W.-F. Wong. Compiler orchestrated prefetching via speculation and predication. In *ASPLOS*, pages 189–198, 2004.

[90] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay. AIFM: High-performance, application-integrated far memory. In *OSDI*, pages 315–332, 2020.

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

[92] D. Shen, J. Luo, F. Dong, X. Guo, K. Wang, and J. C. S. Lui. Distributed and optimal rdma resource scheduling in shared data center networks. In *INFOCOM*, pages 606–615, 2020.

[93] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *MICRO*, pages 42–53, 2000.

[94] A. Shieh, S. Kandula, A. Greenberg, and C. Kim. Seawall: Performance isolation for cloud datacenter networks. In *HotCloud*, 2010.

[95] V. Shrivastav, A. Valadarsky, H. Ballani, P. Costa, K. S. Lee, H. Wang, R. Agarwal, and H. Weatherspoon. Shoal: A network architecture for disaggregated racks. In *NSDI*, pages 255–270, 2019.

[96] D. Shue, M. J. Freedman, and A. Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *OSDI*, pages 349–362, 2012.

[97] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. IOFlow: A software-defined storage architecture. In *SOSP*, pages 182–196, 2013.

[98] Tien-Fu Chen and Jean-Loup Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44(5):609–623, 1995.

[99] S.-Y. Tsai and Y. Zhang. LITE kernel RDMA support for datacenter applications. In *SOSP*, pages 306–324, 2017.

[100] S. P. Vander Wiel and D. J. Lilja. When caches aren't enough: data prefetching techniques. *Computer*, 30(7):23–30, 1997.

[101] S. P. Vander Wiel and D. J. Lilja. A compiler-assisted data prefetch controller. In *Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 372–377, 1999.

[102] G. M. Voelker, E. J. Anderson, T. Kimbrel, M. J. Feeley, J. S. Chase, A. R. Karlin, and H. M. Levy. Implementing cooperative prefetching and caching in a globally-managed memory system. In *SIGMETRICS*, pages 33–43, 1998.

[103] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: Performance insulation for shared storage servers. In *FAST*, 2007.

[104] 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.

[105] 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*, pages 35–53, 2022.

[106] X. Wang and J. F. Martínez. ReBudget: Trading off efficiency vs. fairness in market-based multicore resource allocation via runtime budget reassignment. In *ASPLOS*, pages 19–32, 2016.

[107] H. Yang, A. Breslow, J. Mars, and L. Tang. Bubble-Flux: Precise online qos management for increased utilization in warehouse scale computers. In *ISCA*, pages 607–618, 2013.

[108] S. Yang, T. Harter, N. Agrawal, S. S. Kowsalya, A. Krishnamurthy, S. Al-Kiswany, R. T. Kaushik, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Split-level i/o scheduling. In *SOSP*, pages 474–489, 2015.

[109] 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.

[110] L. Zhang. A new architecture for packet switching network protocols. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE, 1989.

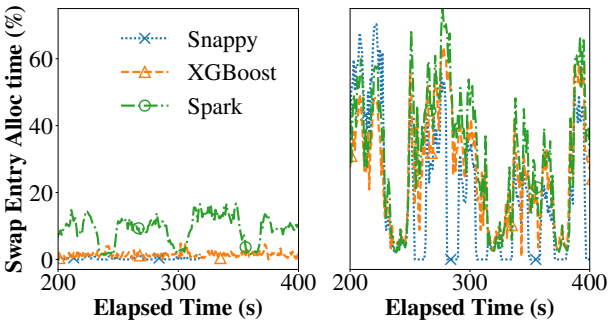
[111] W. Zhang, S. Rajasekaran, S. Duan, T. Wood, and M. Zhuy. Minimizing interference and maximizing progress for hadoop virtual machines. *SIGMETRICS Perform. Eval. Rev.*, 42(4):62–71, 2015.

[112] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. CPI^2: CPU performance isolation for shared compute clusters. In *EuroSys*, pages 379–391, 2013.

[113] Y. Zhang, Y. Tan, B. E. Stephens, and M. Chowdhury. RDMA performance isolation with justitia. In *NSDI*, 2022.

[114] D. F. Zucker, R. B. Lee, and M. J. Flynn. Hardware and software cache prefetching techniques for MPEG benchmarks. *IEEE Transactions on Circuits and Systems for Video Technology*, 10(5):782–796, 2000.

A Extended Motivation



(a) Running individually. (b) Corunning.

Figure 15: Percentage of time spent on swap entry allocation when applications run individually (a) and together (b).

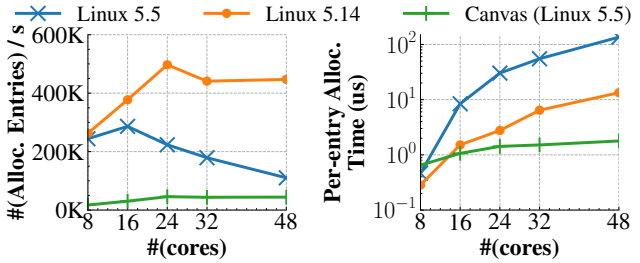
Figure 15 compares the percentage of time spent on swap entry allocation between individual runs and co-runs under Linux 5.5. As shown, each application, when co-run with other applications, spend significantly more time on allocating swap entries due to the increased locking time.

B Recent Kernel Development

As an optimization in Linux 5.5, the kernel keeps swap entries for clean pages—when clean pages are evicted, they do not need to be written back if their swap entries are not released for other allocations. Once a page becomes dirty, its swap entry must be immediately released. Clearly, this approach works for read-intensive applications where most pages are clean, but not for write-intensive workloads such as Spark. We tried various entry-keeping thresholds (*i.e.*, entry keeping starts when the percentage of available swap entries exceeds this threshold) between 25% and 75%, and saw only marginal performance differences (<5%) across our programs.

We have closely followed the kernel development since the release of Linux 5.5 and found two recent patches related to our approach. These two patches, submitted by Intel and merged into the kernel at 5.8, also attempt to optimize locking overhead at swap entry allocation. The idea of the first patch [48] is using fine-grained locking—dividing swap entries into *clusters* and assigning each core a random cluster upon an allocation request. The second patch [46] performs batch entry allocation by scanning more swap entries while holding the lock to make each batch larger. Note that our adaptive allocation algorithm solves a much bigger problem than these patches—Canvas *avoids* allocating entries for most swap-outs, while these patches reduce the overhead of locking for each allocation. As such, Canvas is completely lock-free for reserved entries while these patches must still go through the allocation path, requiring locking if multiple cores are assigned the same cluster (*i.e.*, core collision).

In fact, the probability of collision increases quickly with the number of cores. As shown below in Figure 16, the allocation performance of these patches degrades super-linearly when the number of cores exceeds 24. Another major drawback is that none of these patches build on isolated swap partitions. Lack of swap partition isolation makes applications search for swap entries globally, which can still result in interference—applications such as Spark can quickly saturate these clusters with all its executor threads, making other applications wait before they can obtain the locks. By reserving entries, our algorithm significantly reduces the number of entry allocation requests (due to entry reusing) and the cost of each allocation (due to reduced lock contention).



(a) Swap entry allocation rate. (b) Per-entry allocation time.

Figure 16: Entry allocation comparison between Canvas and the allocation algorithm when Memcached runs on Linux 5.14 on RAMDisk.

Comparison with Linux 5.5 and Linux 5.14. As the kernel is fast evolving and our latest InfiniBand driver is only compatible with Linux 5.5, we compared the swap-entry allocation performance between Canvas, Linux 5.5, and the latest Linux 5.14 over RAMDisk, by running Memcached with varying (8 – 48) cores.

As Figure 16(a) shows, our adaptive entry reservation algorithm reduces the allocation rate by several orders of magnitude compared to Linux 5.14. Note that the allocation rate under Linux 5.5 drops as the number cores increases because each allocation takes much longer and hence the swap-out throughput (*i.e.*, allocation throughput) reduces (*i.e.*, the application runs slower).

Figure 16(b) compares our algorithm with Linux 5.5 and Linux 5.14 on per-entry allocation time. As shown, the optimization in [48, 46] is unscalable—as the number of cores increases, the per-entry allocation cost increases significantly. In fact, the allocation cost grows superlinearly after 24 cores due to core collision. On the contrary, Canvas's per-entry allocation cost remains low and stable. With 48 cores, our algorithm outperforms Linux 5.14's entry allocator (that uses [48, 46]) by **13**×.