RackMem: A Tailored Caching Layer for Rack Scale Computing

Changyeon Jo, Hyunik Kim, Hexiang Geng, Bernhard Egger
Seoul National University
Seoul, Republic of Korea
{changyeon,hyunik,hexiang,bernhard}@csap.snu.ac.kr

ABSTRACT

High-performance computing (HPC) clusters suffer from an overall low memory utilization that is caused by the node-centric memory allocation combined with the variable memory requirements of HPC workloads. The recent provisioning of nodes with terabytes of memory to accommodate workloads with extreme peak memory requirements further exacerbates the problem. Memory disaggregation is viewed as a promising remedy to increase overall resource utilization and enable cost-effective up-scaling and efficient operation of HPC clusters, however, the overhead of demand paging in virtual memory management has so far hindered performant implementations. To overcome these limitations, this work presents RackMem, an efficient implementation of disaggregated memory for rack scale computing. RackMem addresses the shortcomings of Linux's demand paging algorithm and automatically adapts to the memory access patterns of individual processes to minimize the inherent overhead of remote memory accesses. Evaluated on a cluster with an Infiniband interconnect, RackMem outperforms the state-of-the-art RDMA implementation and Linux's virtual memory paging by a significant margin. RackMem's custom demand paging implementation achieves a tail latency that is two orders of magnitude better than that of the Linux kernel. Compared to the state-of-the-art remote paging solution, RackMem achieves a 28% higher throughput and a 44% lower tail latency for a wide variety of real-world workloads.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; • **Software and its engineering → Virtual memory**; **Distributed memory**; *Cloud computing*.

KEYWORDS

Resource disaggregation; Remote memory; High-performance computing; Virtualization

ACM Reference Format:
Changyeon Jo, Hyunik Kim, Hexiang Geng, Bernhard Egger. 2020. RackMem: A Tailored Caching Layer for Rack Scale Computing. In *Proceedings of the 2020 International Conference on Parallel Architectures and Compilation Techniques (PACT '20), October 3–7, 2020, Virtual Event, GA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3410463.3414643

PACT '20, October 3–7, 2020, Virtual Event, GA, USA

ACM ISBN 978-1-4503-8075-1/20/10…$15.00
https://doi.org/10.1145/3410463.3414643

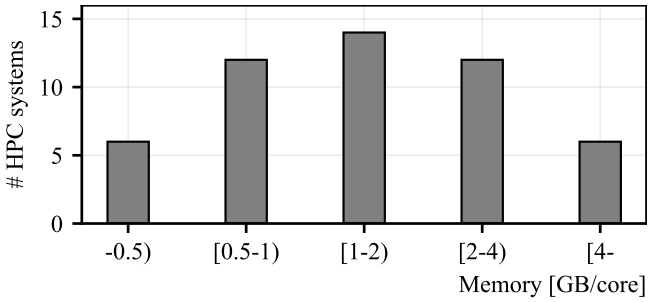

Figure 1: Average memory per core of the fastest 50 systems[1] from the TOP500 list (June 2020) [51].

1 INTRODUCTION

Recent years have brought an increasing demand for applications in the parallel and high-performance computing (HPC) domain. Applications not only comprise core computer science workloads such as in-memory databases or machine learning, but extend across a broad range of science and engineering disciplines such as bioinformatics, climate science, material science, and high-energy physics. The memory system of an HPC cluster plays an important role in accommodating these workloads in terms of performance, cost, and energy consumption [54]. Studies analyzing the memory utilization of HPC clusters find three important characteristics: first, HPC workloads exhibit a bimodal distribution in memory capacity requirements with several workloads requiring over 4 GB per core [54]. Second, the memory usage over time across the nodes in an HPC system shows a large variation ranging from a few hundreds of megabytes up to tens of gigabytes [32]. Third, the working set of a workload is typically significantly smaller than its peak memory requirements [9]. These characteristics make it difficult to determine the 'optimal' memory size in an HPC cluster (Figure 1), leading to low average utilization of the available resources [9, 20].

Recent advances in high-speed interconnects [38] allow for a paradigm shift away from isolated to disaggregated hardware resources [11, 31, 33, 42, 44, 50]. Sharing resources such as processors, memory, or storage over a fast network [1, 6, 24, 29, 30, 48] has the potential to improve resource utilization through flexible allocation that is not possible in server-centric architectures.

Memory disaggregation has been proposed to process big data and in-memory workloads on commodity servers with moderate amounts of physical memory [2, 20]. Disaggregating a low-latency high-throughput resource such as DRAM over a network, however, is a challenging task. Despite improvements in fast optical networks, access latency and throughput are still one to two orders of magnitude below that of local memory [2, 38]. Exploiting the principle of locality, existing implementations employ the local memory as a cache for remote memory. Remote memory is

[1]The fastest 50 systems providing memory information.

implemented by a custom block device and made available to the operating system as a fast swap device [16, 18]; however, Linux's paging subsystem is not optimized for such fast backends. Efforts to improve its performance [34–36] still incur a significant tail latency.

In this work, we present RackMem, a novel RDMA-backed caching layer for rack-scale memory disaggregation under Linux. To avoid the extreme tail latencies of the Linux kernel's default page fault handler, RackMem includes its own paging module that improves the tail latency by one to two orders of magnitude. RackMem exposes memory management policies and access statistics to user space, allowing for an easy integration of custom cache management policies tailored to specific workloads. Implemented in a recent Linux kernel and evaluated on a rack comprising eight compute nodes connected by an InfiniBand interconnect, experiments with various real-world parallel workloads in memory-constrained environments demonstrate that RackMem is able to raise the average memory utilization by 30% and 6% while reducing the total turnaround time by 20% and 8% compared to isolated compute nodes with paging to local SSD and a RDMA-based swap device [16], respectively. RackMem's optimized page fault handler achieves a 97.7% and 98.2% lower 99^{th} percentile latency compared to Linux's paging subsystem backed by local memory and RDMA, demonstrating that RackMem is particularly well suited for latency critical workloads.

The remainder of this paper is organized as follows. Section 2 explains the motivation for this work and analyzes the performance of the Linux page fault handler. Sections 3 and 4 describe the design and implementation of RackMem, its performance is compared against local memory, RDMA, and SSD-based paging devices in Section 5. Sections 6 and 7 discuss related work and conclude this paper.

2 BACKGROUND AND MOTIVATION

2.1 Memory Provisioning in HPC Data Centers

Memory provisioning contributes significantly to the total cost of ownership (TCO) of a HPC cluster [45, 49]. Figure 1 revealed that the fastest supercomputers exhibit a significant variance in the amount of memory provisioned per core. While some of this variance can be explained by different architectures containing specialized accelerators, memory provisioning in HPC clusters is surprisingly ad-hoc with little analysis-based guidelines. Studies have shown that production HPC workloads exhibit a bimodal memory distribution [54], that memory usage varies significantly between nodes in a cluster [32], and that the working set, i.e., the amount of memory accessed during a certain time interval, is typically significantly smaller than the peak memory requirements of a workload. Data center traces by Google in 2011 [17] and Alibaba in 2018 [3] confirm that the average aggregated memory utilization has not improved significantly in that time span and still only reaches about 60 percent [47]. The problem is further aggravated by provisioning up-scaled nodes with many processor cores and terabytes of memory to accommodate new types of workloads with large resource requirements [5, 8, 46]. As a result, this compute-node-centric memory allocation in HPC data centers precludes a high resource utilization and ultimately leads to an increased TCO.

2.2 Resource Disaggregation to the Rescue

Resource disaggregation is a promising approach to overcome the limitations of server-centric models by pooling resources from separate physical machines and making them accessible over a fast network [11]. Figure 2 demonstrates the the potential of memory disaggregation in comparison with the node-centric model. The figure plots the simulated execution a job queue containing 1000 data center workloads (Table 1) on a cluster comprising thirty-five 40-core machines with 16 GB of RAM and five nodes with 20 cores and 32 GB of physical memory each. A job requires between 1-8 cores and 4-32 GB of memory. In the `Node-centric` scenario, a job is placed only if a node can satisfy the requested CPU and memory resources. The `Optimal` scenario shows the potential (and upper limit) of memory disaggregation by assuming no performance penalty for remote memory accesses. This scenario is logically identical to pooling the resources of all 40 nodes into a single node with 1500 CPU cores and 720 GB of memory. The scenarios `RackMem` and `Infiniswap` show the performance of the presented approach and the state-of-the-art RDMA paging [18]. Jobs are placed on a node if at least 50 percent of the requested memory resources are available locally. In these scenarios, remote memory accesses incur overhead; the slowdown in dependence of the amount of available local memory is obtained from real-world experiments (Figure 7).

The cumulative distribution function (CDF) of the job completion time for the four scenarios is shown in Figure 2 (a). The mean turnaround time of `Optimal`, `RackMem`, `Infiniswap`, and `Node-centric` is 48.4, 59.1, 100.2, and 120.5 epochs, respectively, and all disaggregation scenarios show clear benefits over the node-centric model even with realistic performance penalties. The aggregated memory utilization is plotted in Figure 2 (b). We observe that that memory disaggregation allows the available resources to be utilized almost fully at all times. The simulation also demonstrates the benefits of RackMem's optimized demand paging in contrast to Inifiniswap which is implemented as a swap device backend to Linux's demand paging system.

2.3 Limitations of Linux's Demand Paging

A common choice to transparently support additional storage is to employ the virtual memory management (VMM) capabilities of commodity hardware. The operating system's page fault handler, invoked when an application tries to access a virtual memory page currently not residing in the local memory, forwards the request to read/write a memory page from/to the storage device. Google's FarMemory [26] and Infiniswap [18] both plug into Linux's page handler by registering as a storage backend. LegoOS [47] implements distributed virtual memory in the page fault handler.

GNU/Linux is the de-facto standard operating system for data centers. Linux has supported demand paging since early on, and its implementation is primarily optimized for block device-based swap devices such as harddisks and, more recently, solid-state drives (SSDs). Linux' implementations of page fault handling and demand paging, while heavily optimized, have grown into complex pieces of code with deep and complex call paths and comprising several thousand lines of code. Inherently optimized for block devices several orders of magnitude slower than main memory, Linux' demand paging does not perform well with fast backends such as Infiniband.

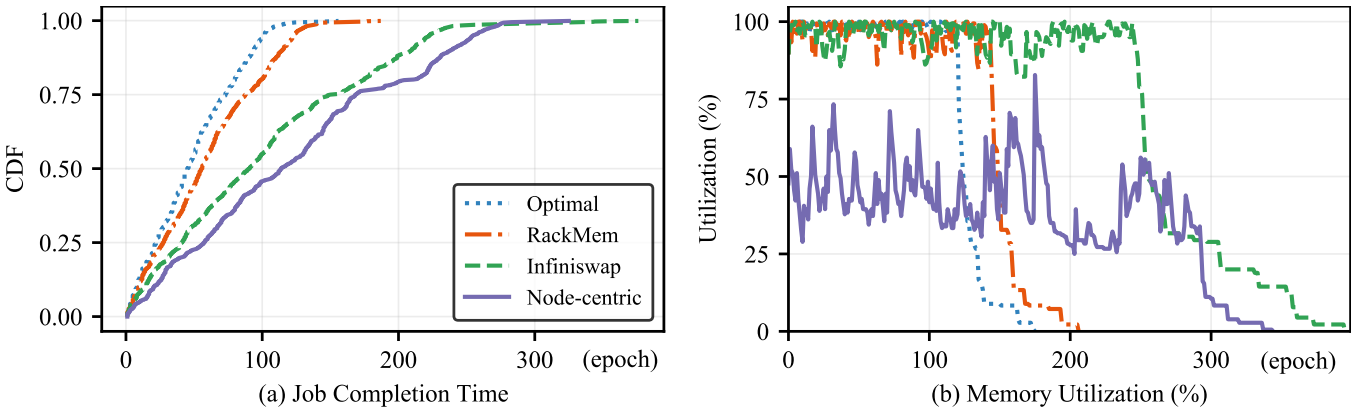

Figure 2: Job completion time and memory utilization of 1000 workloads for different resource sharing models.

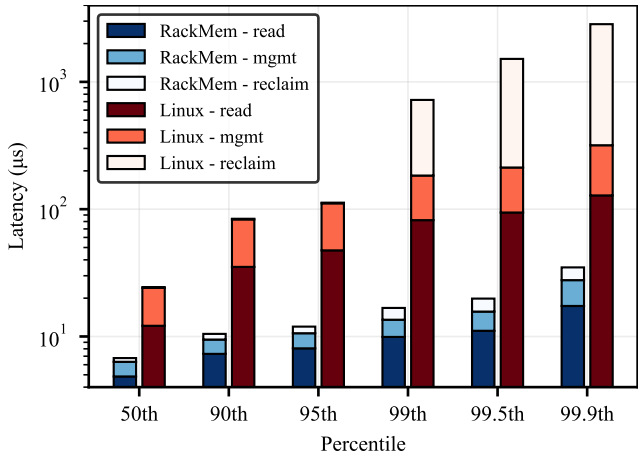

Figure 3: Log-scale pagefault handler latency in Linux v5.3.9. The latency breakdown is for `do_swap_page` in `/mm/memory.c`.

Figure 3 plots the latency of Linux' pagefault handler (kernel version 5.3.9) with an RDMA backend over Infiniband and RackMem's tailored page fault handling for different percentiles. To stress demand paging, the Spark PageRank benchmark is executed in a cgroup limiting the available local memory to 30% of the workload's working set size. Looking at the performance of the Linux pagefault handler first, compared to the median value (50^{th} percentile) with a latency of $24\mu s$, the 99^{th} and 99.9^{th} percentile exhibit a severe 43- and 81-fold slowdown at $721\mu s$ and $2840\mu s$, respectively.

Handling a demand paging pagefault can be broken down into three actions: `mgmt`: management overhead caused by updating data structures and page tables, `read`: reading a page from backing storage, and `reclaim`: reclaiming a page in local memory[2]. The bars in Figure 3 reveal the latencies of the three actions. While `mgmt` and `read` observe a "modest" 10-fold slowdown from the 50 to the 99.9^{th} percentile, the slowdown of `reclaim` is dramatic: 167-fold at 99, 313-fold at 99.5, and 351-fold at the 99.9^{th} percentile. The analysis of Linux's pagefault handler reveals four bottlenecks that hamper exploiting the full potential of fast disaggregated memory.

[2]Page reclamation involves finding a victim page and can trigger writing data to backing storage if no free page is available

1. Contention on shared data structures. All processes share a global memory pool that are managed in a least-recently-used (LRU) queue of accessed memory pages. This data structure becomes the main bottleneck when the system is frequently paging in/out data from/to fast paging devices such remote memory. The synchronous page reclamation process' accesses to the LRU queue and other shared data structures in particular is one of the main culprits for the extreme tail latency.

2. Tail-latency of page reclamation. The pagefault handler in Linux frequently triggers the page reclamation function when there is no readily available free page. This operation can block a pagefault for a few milliseconds and has been identified to be the main performance bottleneck of under intensive paging. To reclaim pages, Linux' pagefault handler scans the entire LRU list in the system to find free pages. The reclaim function has a deep call depth and nested loops inside and takes up to $2600\mu s$ to complete - which is a few hundred times slower than a remote page access over Infiniband RDMA. While this degree of latency can be hidden and is acceptable for slower storage backends, it is not suitable for the fast storage backends available today.

3. Slow data path. The mismatch between the block device abstraction and the interface for remote memory adds significant overhead to the data path. Bypassing the block device I/O stack by directly invoking the remote memory APIs can significantly reduce the access latency to remote memory.

4. Lack of workload-specific optimizations. All processes are treated equally by Linux's virtual memory implementation. This design prevents fine-tuning paging policies for applications with distinct memory access patterns to further reduce the overhead of memory disaggregation.

Without significant modifications to the demand paging implementation in Linux, is is impossible to avoid this tail-latency problem for to virtual memory backed by fast remote memory [18, 37]. Modifying Linux' demand paging implementation comprising a few tens of thousands of lines with complex call paths is a challenging task that is not only likely to introduce new bugs but could also lead to security problems. The approach chosen by RackMem bypasses Linux's demand paging altogether and implements an custom page fault handler that is optimized for parallel access, fast storage backends and tailored for RDMA over Infiniband. The following sections describe the design and implementation of RackMem in detail.

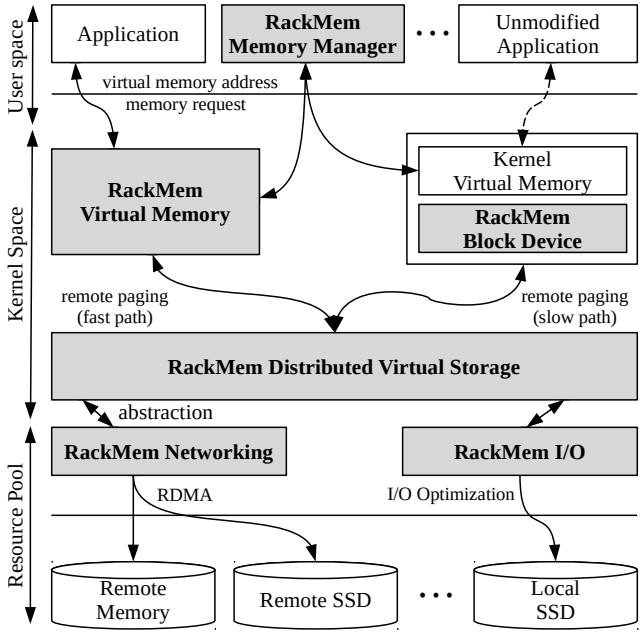

Figure 4: High-level architecture of RackMem.

```
def rackmem_vm_fault(struct vm_fault *vmf)
{
    /* Step 1: obtain pointer to RackMem region */
    rr = (struct rack_region *) vmf->vma->private_data;
    /* Step 2: lock page */
    r_page = &rr->pages[index];
    lock(&r_page->lock);
    /* Step 3: page still free list in local memory? */
    if (r_page->flags & INACTIVE)) { /* yes */
        /* set flag for prefetched pages */
        if (r_page->flags & PREFETCH) cache_hit = 1;
        /* remap page and move back to active list */
        rack_page_remap(rr, r_page, address);
    } else { /* no */
      /* Step 4: get free page to hold faulted page */
      rack_get_page(rr, &r_page->buf);
      /* Step 5: restore page from backend storage */
      rack_page_restore(rr, r_page);
    }
    /* Step 6: remove lock */
    unlock(&r_page->lock);
    /* Step 7: (optional) proactive page reclamation */
    rack_request_reclamation(rr, nr_pages);
    /* Step 8: (optional) prefetch */
    rack_prefetch_pages(rr, address, window_size, cache_hit);
}
```

Figure 5: Code of the RackMem pagefault handler.

3 RACKMEM DESIGN

From an operational viewpoint, RackMem utilizes the local memory as a cache for distributed memory. Each memory region managed by RackMem has a (dynamically adjustable) limit on how much local memory it is allowed to occupy; data not residing in the local cache is located on a remote node.

Figure 4 illustrates the high-level architecture of RackMem. Grey boxes represent components implemented by RackMem, white boxes show applications, kernel components, or storage devices not specific to RackMem. The first of two core components is the **Distributed Virtual Storage** (DVS) layer. DVS provides a reliable, distributed, byte-addressable storage interface to different local and remote storage types. The simple interface allows applications or other kernel components to access unused memory located in remote nodes. DVS provides optional reliability through replication and policies to utilize idle memory resources in the cluster in a balanced manner. Access to different types of backends is provided by lean drivers that expose a uniform interface to the DVS layer for allocation, deallocation, and I/O. In Figure 4, the components **RackMem Networking** and **RackMem I/O** provide access to remote memory over RDMA and remote and local file storage; additional types can be added by implementing the interface for the new backend.

The second core component of RackMem is the **Virtual Memory** layer. This layer provides independent and highly-optimized dynamic paging for high performance backends such as RDMA, called the "fast path". User-space applications first register RackMem-managed virtual memory regions through a system-call interface after which native memory read/write instructions are able to access distributed storage in a transparent manner. The main disadvantage of this design is that applications need to be modified to make use of the optimized RackMem virtual memory. For virtual machines, seamless integration is provided by a modified hypervisor; for applications that cannot be modified, RackMem supports remote paging via Linux's virtual memory implementation (the "slow path") by providing the **RackMem Block Device**. The **RackMem Memory Manager**, finally, is a user-space component offering advanced memory management features such as collecting detailed statistics, dynamically repartitioning the amount of available local memory, or implementing advanced application-specific memory management policies for the different RackMem memory regions. Other user-level components of RackMem include a naming service and a discovery protocol and are not shown in Figure 4.

3.1 RackMem Virtual Memory

The main purpose of RackMem Virtual Memory (RVM) is to provide an abstraction for remote memory. Commodity hardware does not (yet) support transparent and direct access to remote memory, thus requiring explicit management of remote memory. The virtual memory abstraction to remote memory implemented in RVM allows for transparent access to remote memory and minimized the transitioning overhead from a server-centric to a disaggregated architecture.

An important design consideration of virtual memory – besides usability – is performance. Since remote memory accesses are still an order of magnitude slower than accesses to local memory [38], the benefit of memory disaggregation is easily lost if disaggregation incurs a significant performance. Small improvements in virtual memory performance translate to millions of dollars at a large scale [26], hence, RackMem's main goal is set to improve virtual memory performance with fast storage backends.

3.1.1 Design Considerations for RVM. The analysis of Linux's demand paging (Section 2.3) revealed severe contention on global shared data structures with fast storage backends. To eliminate this bottleneck and improve scalability, RackMem maintains separate data structures for each virtual address space.

Virtual memory management. Requests for virtual memory managed by RackMem are handled by RVM. For each request, RVM creates a unique descriptor and initialized the necessary data structures. RackMem transparently manages its virtual address ranges; hardware pagefaults are handled directly by RVM, hence, user-level applications can access the entire address region using native memory operations. Memory is managed at page granularity and the actual allocation is delayed until the first access. When an application first accesses an unmapped page, a new page is allocated in the local memory cache and mapped into the application's virtual memory space. If the local cache is full, one or more vicitm pages are paged out to backend storage via the DVS layer before a new page is allocated or a previously paged-out page is brought back. The cache size of each RackMem memory region can be dynamically adjusted through a system call interface from userspace.

Pagefault handling. While pagefault handling latency is a concern for any demand paging system, the adverse effects of a inefficient pagefault handler are more severe with fast backends. RackMem uses a number of techniques to reduce the number of pagefaults and the latency of its pagefault handler to a minimum. First, separate weakly-ordered active and free lists of pages are maintained for each region. Second, RackMem actively attempts to reduce the number of faults through prefetching. Third, it seeks to always keep a minimum number pages on the free list to avoid the time-consuming operation of selecting a victim page and paging out its contents to remote storage before the faulted-on page can be brought back in. Figure 5 shows the annotated code of RackMem's fast pagefault handler that proceeds with the following steps.

(1) Obtain pointer to data structures of affected region.
(2) Lock page to prevent data corruption.
(3) Lookup page in local cache. If found, map page into virtual address space and goto step 6.
(4) Find a free page in cache; if no such page exists, select and evict victim page.
(5) If accessed page has been paged out, restore its contents.
(6) Unlock page
(7) Wake-up proactive page reclamation thread to proactively evict infrequently accessed pages and create a pool of available free pages.
(8) Perform synchronous prefetch with data around the faulting address to reduce future pagefault handling overhead.

3.1.2 Latency hiding techniques. This section discusses the two main latency hiding techniques, prefetching and proactive page reclamation, in more detail.

Prefetching. Prefetching data to local memory ahead of an access significantly reduces the pagefault handling latency by eliminating costly I/O operations during pagefault handling. Especially batch data processing applications frequently exhibit sequential memory access patterns for which prefetching is particularly effective [21, 22]. The separation of data structures per memory region allows RackMem to track multiple access streams for co-located applications; heuristics that exploit this information are discussed in Section 4.1.

Reactive and proactive page reclamation. When the local cache is full, accessing an unmapped page requires selecting and paging out of a victim page. Page reclamation faces three challenges: first, if the victim page is dirty its contents have to be paged out which incurs a significant overhead. Second, selecting the "wrong" vicitm page, i.e., a page that is accessed soon again, will severely hurt performance. Third, I/O over modern RDMA interconnects is so fast that the latency of a sophisticated victim selection algorithm easily offsets the benefits of accurate victim selection. RackMem addresses these challenges by employing both reactive and proactive page reclamation policies. Reactive page reclamation occurs when the cache is completely full and a page needs to be reclaimed to serve a pagefault. This overhead cannot be hidden, so victim selection should be fast rather than accurate. Proactive page reclamation, on the other hand, reduces the chance of reactive page reclamation by proactively evicting less-frequently accessed data and placing the pages on the free list. This mechanism is implemented in a background task that does not directly affect the latency of the pagefault handler; a more accurate and computationally intensive victim selection can thus be employed. Implementations of the two page reclamation techniques are discussed in Section 4.1.

3.2 RackMem Distributed Virtual Storage

RackMem's DVS layer abstracts the backend storage devices and presents a combined linear byte-addressable address space to the upper layers. DVS supports not only remote memory as backend storage devices; any device that implements the backend DVS API can be used by DVS. Virtual addresses are translated into backend device locations comprising of the backend ID and the memory offset in the storage device.

The DVS layer handles selection of the backend storage devices, plus storage allocation and reclamation. DVS does not identify remote or local storage devices, instead, APIs to register backend storage devices and to inform the DVS layer about the amount of free storage on each backend device are provided. In the current implementation, the tasks of identification, naming, and periodically reporting the status of each backend device are implemented by user-level tasks.

DVS supports optional fault tolerance and replication by duplicating data to a second remote node or local storage.

4 IMPLEMENTATION

RackMem's virtual memory layer, the distributed virtual storage layer, and the remote storage backends are implemented by separate Linux kernel modules. RackMem exposes the distributed storage via the character device `/dev/rackmem` and the management API through a `debugfs` device. This implementation allows users of RackMem to allocate distributed storage with a call to `mmap()` and to manage and tune the operation of RackMem through simple Unix file I/O on the `debugfs` device. In the remainder of this section, implementation details of the different parts of RackMem are discussed.

4.1 RackMem Virtual Memory Module

The RackMem VM (RVM) module registers the character device `/dev/rackmem`. An application requests a RackMem virtual memory region by invoking the `mmap()` system call with a file descriptor to that device and the desired size of the the region. RVM registers its own pagefault handler to process pagefaults occurring in the newly allocated region as described in Section 3.1.1.

Prefetch algorithm. The prefetch algorithm in RVM implements a modified version the virtual memory area-based (VMA-based) prefetching technique that has recently been integrated into the Linux kernel. VMA-based prefetching exploits spatial locality in the virtual address space rather than the physical location of the data. VMA-based prefetching achieves better accuracy than physical address-based prefetching at the expense of losing the ability to exploit sequential I/O requests. This is, however, less of a concern since recent storage devices such as SSDs, NVMe, or RDMA-backed remote memory provide good random access performance.

RackMem maintains a dynamic window size that dictates how many pages are prefetched after a pagefault. Other than the VMA-based prefetcher, RackMem tracks pagefaults on a *per-core*-basis by storing the prefetch window size and the last faulted address in processor-local storage. This simple optimization allows RackMem to detect and exploit distinct per-core memory access patterns.

The synchronous prefetching is invoked at the end of a pagefault (Figure 5). In our experiments, asynchronous prefetching in the background did not yield significant performance benefits. The reason for this somewhat unexpected result is that the latency of a prefetch operations is lower than the overhead incurred of asynchronous I/O and extra synchronization in a background thread.

Active and free list. Pages mapped into the virtual memory space of a user application are maintained on the so-called *active* list, whereas available (unmapped) pages are included in the *free* list. These lists are managed separately for each RackMem memory region. A page is added to the tail of the active list (= most recently inserted) when it is mapped into the virtual address space in reaction to a pagefault. A page that is proactively paged out is moved to the free list but keeps its data so that it can be brought back quickly if referenced before being paged out. The data structure of a page contains an 8-bit value that represents a metric for the its access time and frequency. A background task periodically scans the active bits of the hardware page tables. Upon updates, the 8-bit value of a page is rotated right by one bit and the access bit of the page is stored in the MSB (most significant bit). Interpreted as an integer, this 8-bit value directly reflects the access time and frequency of the page. Note that the position of pages on the active list is determined entirely by the insertion order, i.e., the active list is not re-ordered based on the access times of the active pages.

Reactive and proactive page reclamation. As discussed in Section 3.1.2, RackMem employs different policies for reactive and proactive page reclamation. If the free list is empty, a page is reactively reclaimed by paging its contents out to remote memory. The latency of reactive page reclamation needs to be a short as possible since it adds to the total pagefault handling latency. RackMem employs a constant-time algorithm that simply selects the page located at the head, i.e., least recently accessed position of the access list as the victim page. To avoid reactive page reclamations, a proactive page reclamation thread is woken-up at the end of the pagefault handler if the number of pages on the free list is lower than a given threshold min_{free}. Since sorting the active list based on the page access frequencies would be too computationally expensive, RackMem employs an approximate LRU algorithm that scans the first $k * min_{free}$ pages from the head of the list (= least recently inserted). A page is selected as a victim if its normalized access score is below threshold t as follows

$$victim(page) = \begin{cases} yes & \text{if } (p - mean) \big/ stdev \leq t \\ no & \text{otherwise} \end{cases}$$

where p stands for the access score of the page, *mean* and *stdev* represent the mean value and standard deviation of all active pages' access scores, and t is the selection threshold. This heuristic avoids sorting the active list and provides a simple way of trading computational overhead for accuracy. We empirically set $min_{free} = 512$, $k = 4$, and $t = 0$.

4.2 RackMem DVS Module

The RackMem DVS kernel module provides byte-addressable disaggregated storage with a set of storage APIs that support allocation, deallocation, and data transfer operations. This section presents the implementation details of DVS.

Memory management. Similar to RVM, DVS regions are identified by a descriptor that created upon request and used to identify the region in subsequent calls to I/O operations. Distributed storage is allocated lazily, i.e., allocation is delayed until the first access. DVS manages distributed storage in fixed-size slabs of 64 MB, a value that was empirically shown to reduce the number of I/O requests to remote storage while providing sufficient granularity. A slab is allocated when its data is accessed for the first time. DVS selects a donor node from the list of available nodes providing backend storage using the "power of two choices" distributed load-balancing algorithm [40] that has shown good results in practice. The algorithm randomly picks two nodes from the node list and selects the one that has more idle memory.

Remote memory and other storage backends. RackMem DVS supports several remote and local storage backends. The RDMA backend is based on LITE [52] and provides a high-performance and easy-to-use networking stack for remote memory-backed storage. RackMem's networking module implements abstractions for remote memory, state sharing, and coordinated I/O polling to provide programmability, scalability for multiple connections, and efficient I/O completion. In addition to the RDMA backend, RackMem also implements local memory-backed storage (that can be used to emulate `pmem` devices), and local file-backed storage. The optional parallel output to local file-backed storage provides redundancy and fault tolerance in case of a failure of a remote storage node.

Userspace memory agent. RackMem employs a userspace memory agent in each node that processes remote memory allocation and deallocation requests. The agents communicate with the storage backends in remote nodes through a custom RPC stack over RDMA. In response to a memory allocation request, the agent allocates a memory block of the requested size and returns the necessary information to allow a remote memory to access the block

through RDMA. Similar to *Infiniswap* [18], the memory agents also monitor and broadcast local memory pressure, and reclaim remote nodes when the memory pressure of the system is too high.

Reliability. RackMem DVS supports optional data replication to remote or local storage backends. DVS performs synchronous write operations to equally fast or faster storage types and employs asynchronous write duplication for slower backup storage. In case of a failure of the main storage backend, DVS redirects the request to the backup storage device. Faults are transparent to the application and only noticeable by the reduced performance caused by the slower backup storage device.

Remote node discovery and naming. Remote nodes are announced (registered) and deregistered through RackMem's `debugfs` interface. The discovery and registration is automated as an extension to the userspace memory agent. The userspace memory agents periodically exchange messages and register newly discovered nodes and deregister nodes that have disappeared. When a node is deregistered in an orderly manner (i.e., did not fail), the DVS module fetches all data from the leaving node and redistributes it to other nodes. Once all remote memory has been moved away from the leaving node, the RackMem software stack can be unloaded.

4.3 Dynamic Rebalancing of Local Memory

RackMem provides a user-level API that allows to dynamically adjustment the local memory cache size of RackMem memory regions. In addition, statistics about RVM and DVS such as the access frequency byte of each page or the total number of pagefaults are exposed to user-space through RackMem's `debugfs` device. This design allows for powerful memory optimizations based on the obtained statistics. As a proof-of-concept, we have implemented a simple local memory rebalancer in Python that periodically rebalances the local cache size of all RackMem memory regions in proportion to the number of active pages and pagefaults of a region. Section 5.6 and Figure 11 discuss the effects of dynamic memory partitioning.

4.4 RackMem for Virtual Machines

The `/dev/rackmem` device node provides a simple interface to utilize RackMem-backed distributed memory, however, applications need to be modified to allocate memory from RackMem. For native applications, library interpositioning is used to intercept calls to the dynamic memory management library and are elaborated in Section 4.5 below. To seamlessly support Infrastructure-as-a-Service (IaaS) virtualization where workloads are executed in isolated virtual machines [2], RackMem includes a modified QEMU/KVM hypervisor. The RackMem QEMU/KVM hypervisor maps the memory of a virtual machine to a RackMem memory region instead of local memory. The modification requires only a few additional lines of code that open RackMem's `/dev/rackmem` device and pass its descriptor to the (already present) `mmap()` system call to setup the memory of a VM.

4.5 Running unmodified applications

Thanks to extensive hardware support and efficient virtual machine monitors, applications run with little performance overhead in a VM. Nevertheless, running applications natively without modifications is an important use case especially for HPC. To provide the fast paging and benefit of remote memory access to native applications, RackMem includes a library that can be interpositioned to intercept the dynamic memory management requests (`malloc/free`, `new/delete`, `mmap/munmap` and variants) of the C standard library. The library creates one RackMem region that serves as the heap area for the numerous and often small dynamic memory requests. This avoids interacting with the lower levels of RackMem and is a design also chosen by related work [1].

4.6 Linux Demand Paging on RackMem

RackMem's DVS layer can also serve as a storage backend to Linux' virtual memory manager (Figure 4). We modify the implementation of Linux's `nullb` high-performance block device [15] to employ RackMem's DVS as its backend storage device. While such an approach forfeits the advantages of RackMem's fast path implemented in its virtual memory module, it enables Linux demand paging to disaggregated memory and thus also unmodified applications to profit from RackMem.

The implementation of Infiniswap [18] as the representative of the state-of-the-art backend to Linux demand paging is also implemented in this way. One of the main performance optimizations of Infiniswap is its use of `nullb`'s per-core I/O request queue. We have further optimized the Infiniband block device by implementing I/O merging and by using the scatter/gather functionality of the RDMA driver. This improved implementation of Infiniswap is used as the main comparison target in the following evaluation section.

5 EVALUATION

RackMem has been evaluated thoroughly on real hardware (Section 5.1) with a wide range of applications (Section 5.2) and a number of scenarios. The remainder of this section answers the following questions:

- Can RackMem's virtual memory improve the pagefault handler throughput and reduce the tail latency? (Section 5.3)
- Does RackMem provide better performance for applications under a disaggregated environment? (Section 5.4)
- How efficiently can RackMem utilize local memory when co-located applications share local memory? (Section 5.6)
- Can RackMem improve performance of a job processing cluster by sharing unused memory between physical machines? (Section 5.7)

5.1 Execution Environment

All evaluations and experiments are performed on a cluster composed of four physical nodes comprising a Xeon Silver 4114 processor (10 cores / 20 threads) and 64GB of DRAM. Each machine uses a Mellanox InfiniBand ConnectX-4 NIC (56 Gbit/s single-port throughput) for RDMA networking and an Intel SSD SC2KB480G7R for local storage. All nodes run Linux 5.3.9 with KVM/QEMU 2.11 for virtual machines.

Effect of RackMem Optimizations. RackMem is evaluated in the four distinct configurations `RACK`, `RACK.R`, `RACK.P`, and `RACK.RP` to evaluate the effect of individual optimization described in Sections 3.1 and 4.1. The details of the different configurations are listed in Table 2.

	Mean (5s)	Max (5s)	Stdev (5s)	Mean (30s)	Max (30s)	Stdev (30s)	Mean (60s)	Max (60s)	Stdev (60s)	Peak RSS
Spark.PgRank	157	1421	323	947	3365	1113	2104	4181	1368	4372
Parsec.Bodytrack	98	1270	211	553	1970	629	1198	2190	787	2234
Parsec.Canneal	62	636	114	382	1289	426	822	1730	593	1845
Parsec.Dedup	213	2696	443	1064	4148	1324	2416	4734	1698	4832
Parsec.Raytrace	94	923	161	579	1696	551	1241	2389	765	2826
Parsec.Vips	175	2023	354	932	3506	1133	2105	4090	1457	4178
NPB.bt	28	752	104	184	1336	373	415	1401	549	1401
NPB.dc	46	754	125	301	1295	416	663	1399	542	1400
NPB.ft	59	1273	196	383	1871	592	881	1984	799	1983
NPB.is	60	878	159	393	1542	523	898	1736	658	1737
NPB.mg	44	536	109	288	1085	372	640	1147	436	1145
OLTPBench.tpcc	93	458	62	259	1318	317	488	1902	549	2715
OLTPBench.twitter	62	368	90	297	1346	407	567	2361	738	3421
OLTPBench.wikipedia	121	638	78	410	1514	292	766	2023	449	5199

Table 1: Working set size for 5/30/60 second windows and peak resident set size (RSS) of target applications. Values in MB.

Configuration	Description
`RACK`	base implementation
`RACK.R`	`RACK` with proactive page reclamation
`RACK.P`	`RACK` with prefetching
`RACK.RP`	`RACK` with prefetching and proactive page reclamation

Table 2: Evaluated RackMem configurations.

Implementation	Pagefault Handler	Backend
`LMEM`	Linux	local memory
`SSD`	Linux	local SSD
`Infiniswap`	Linux	RDMA distributed memory
`RackMem`	RackMem	RDMA distributed memory

Table 3: Compared implementations.

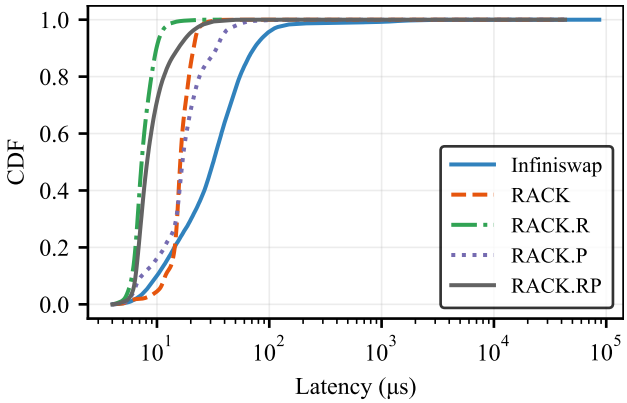

Figure 6: Pagefault handling overhead of RackMem.

Targets for comparison. RackMem is compared against two different local backends and the state-of-the-art RDMA distributed memory implementation from related work. Table 3 gives an overview of the evaluated implementations. `LMEM` stores and retrieves data in the local DRAM. This approach uses `pmem.io` [43] to avoid the block layer overhead of a RAM disk. `SSD`, on the other hand, evaluates Linux virtual memory backed by a local SSD. `Infiniswap` represents the state-of-the-art of Linux virtual memory paging to a RDMA-backed distributed storage backend. `Infiniswap` is the main competitor and comparison target to `RackMem`. For the comparison, we use an improved implementation of Infiniswap (Section 4.6) because the open-sourced implementation [19] does not run correctly on recent Linux kernels which also include patches that improve the performance of the swap subsystem [12].

Enforcing local memory limits and sharing. To evaluate performance with a varying local cache size, we artificially limit the amount of local memory available to the applications. In the case of RackMem, the amount of local cache size can be dynamically set through the `debugfs` interface (Section 4). In scenarios that employ Linux's virtual memory with demand paging (`LMEM`, `SSD`, and `Infiniswap`), the amount of memory available to the workload is limited through the use of `cgroups` [13] by setting the `memory_limits_in_bytes` parameter to the desired value.

5.2 Target Applications

RackMem, Infiniswap, and the Linux-native targets for comparison are evaluated with a wide range of real-world workloads including batch-oriented and latency-critical applications. Batch-oriented applications include PageRank from Spark (`PgRank`) [53], CIFAR10 inference from Tensorflow (`Tensorflow.CIFAR10`), four applications (`Canneal`, `Raytrace`, `Vips`, and `Dedup`) from the Parsec benchmark suite [7], and five workloads (`mg`, `is`, `ft`, `bt`, and `dc`) from the NASA Parallel Benchmarks (NPB) [4]. The applications `tpcc`, `twitter`, and `wikipedia` from OLTP-Bench [10] represent latency critical workloads.

Table 1 lists the benchmarks along with the mean, the maximum, and the standard deviation of the working set size for a sliding window of 5, 30, and 60 seconds, respectively. The last column shows the peak resident set size (RSS), i.e., the highest amount of memory allocated over the entire course of execution for each application.

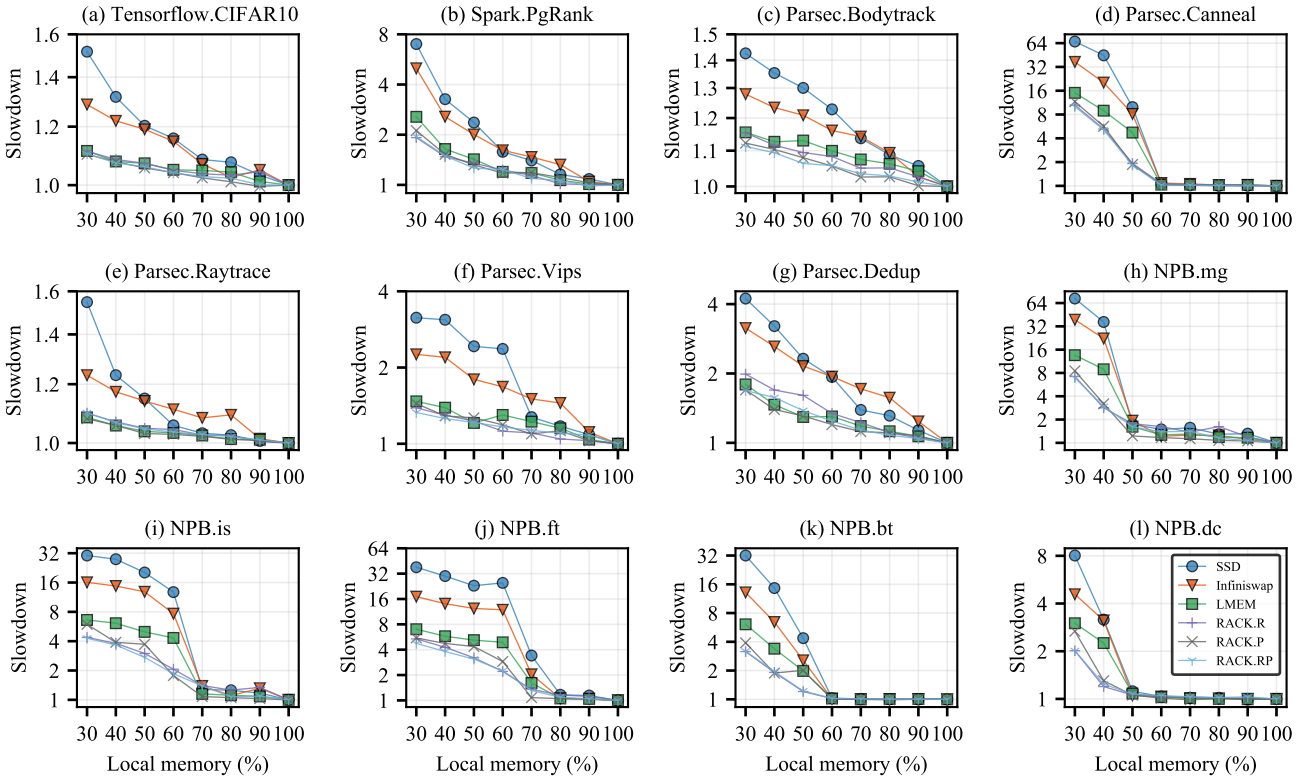

Figure 7: Normalized execution time of batch-oriented applications under memory limits (log scale, lower is better).

5.3 Pagefault Handler Latency

The latency of handling major pagefaults by RackMem and the Linux kernel is measured by adding tracepoints [14] to the relevant functions in the Linux kernel and the virtual memory module of RackMem. The PageRank benchmark `Spark.PgRank` is executed in a VM with a local memory limit of 30% of its peak RSS.

The breakdown of the latencies has already been shown in Figure 3. That figure revealed the extreme tail latencies of Linux demand paging with fast backends. RackMem, on the other hand, not only achieves a 4-fold shorter latency at the 50^{th} percentile, but its tail latency increases only modestly from 7μs at 50, 12 at 95, 17 at 99, and 35μs at the 99.9 percentile. Compared to Linux, RackMem achieves an 80-times shorter latency at the 99.9 percentile.

Figure 6 plots the cumulative distribution function (CDF) of the pagefault handling latency Infiniswap and RackMem with different optimizations enabled. The results clearly show the benefit of RackMem's latency-optimized pagefault handler compared to Linux's implementation. All configurations of RackMem significantly outperform Infiniswap using the Linux pagefault handler, both for the median latency and especially also in terms of tail latency. Compared to the median latency of 24.4 µs in the case of Linux/Infiniswap, the median latency of RackMem with reactive page reclamation is over three times smaller at 6.76 µs.

The figure also visualizes the impact of proactive reclamation and prefetching on latency. `RACK.R` and `RACK.RP` both exhibit shorter latencies than RackMem without proactive page reclamation, demonstrating the benefit of avoiding victim page selection and eviction on the critical path. Note that prefetching not only leads to to shorter latencies but also increases tail latency compared to no prefetching. This is visible in the CDF from Figure 6: Almost all pagefaults under RackMem without optimizations (`RACK`) exhibit latencies between 10 and 20 μs. With prefetching (`RACK.P`), the fastest 25 percent of pagefaults experience a shorter latency, and the 40 percent of the tail latencies are longer compared to `RACK` (note the crossover points of the two policies in the CDF graph). The reason for this behavior is as follows. Prefetched pages are not directly mapped, but added to the free list. If a fault occurs on such a page (prefetch hit), the page is moved to the active list and mapped into the process' address space without causing I/O operations, yielding a larger number of pagefaults with short latencies. The longer tail, on the other hand, is caused by prefetching itself which is performed synchronously at the end of the pagefault handler (Figure 5).

5.4 Single Application Performance

This section analyzes how RackMem improves end-to-end application performance for batch-oriented and latency-critical workloads.

5.4.1 Batch-oriented Applications. The sensitivity of an application with regards to memory disaggregation is determined by measuring the relative slowdown while restricting the available local memory from 100 percent (baseline, local performance) down to 30 percent.

Figure 7 plots the normalized execution times of the batch-oriented applications under the different configurations. The results demonstrate that RackMem's optimized virtual memory directly

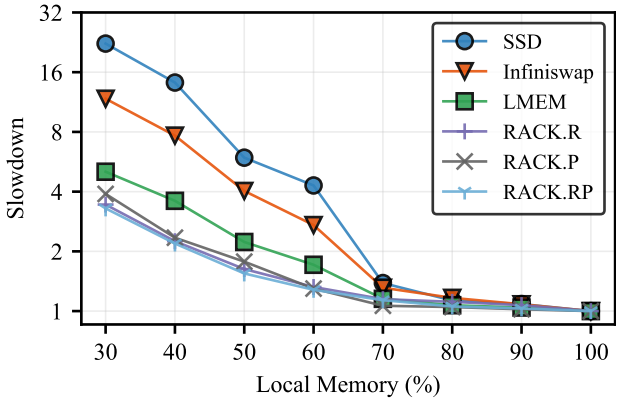

Figure 8: Averaged normalized execution time of all batch-oriented applications.

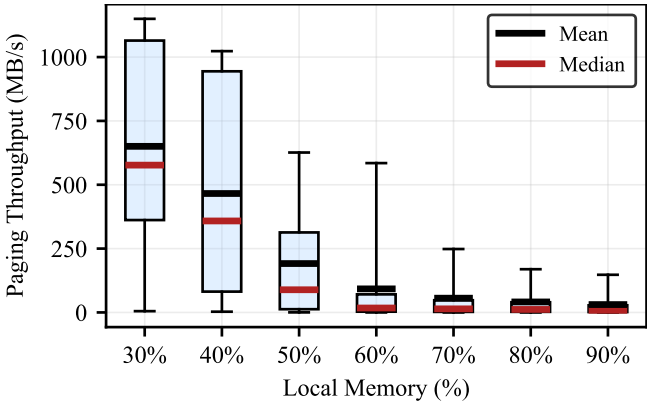

Figure 9: RDMA NIC utilization of RackMem.

translates to end-to-end application performance gains under memory disaggregation. Interestingly, for most applications RackMem even outperforms `LMEM` that uses the much faster local memory as a paging device. This result again implies that Linux's virtual memory implementation becomes the main bottleneck for fast storage devices. `SSD` and `Infiniswap` show a lower tolerance under intensive paging scenarios, and the benefit of utilizing additional memory is lost by the severe performance degradation.

Looking at the effect of RackMem's page reclamation and prefetch optimization in Figure 7, in most cases, the benefits are not clearly visible. This implies that RackMem's efficient implementation of virtual memory contributes the most to the performance improvement. The results in Figure 7 also reveal that the proactive page reclamation optimization clearly improves application performance under intensive disaggregation scenarios and that the prefetch optimization contributes more than proactive reclamation when applications exhibit sequential memory access patterns such as in `Parsec.Dedup` and `Parsec.Raytrace`. The aggregated results over all batch-oriented applications in Figure 8 show the benefits of RackMem over the other approaches and the effect of RackMem's optimizations. At 30% local memory, the average normalized execution time for `RACK.RP`, `RACK.R`, `RACK.P` are 3.29, 3.44, and 3.89, respectively, demonstrating that both individual optimizations contribute to `RACK.RP`, the best performing configuration.

5.4.2 Latency-oriented Applications. For latency-oriented applications, we measure the transaction latencies of the OLTP workloads at 30% of the peak RSS available as local memory. Figure 10 plots the function $1 - CDF$ in log scale to effectively visualize the tail latency. We compare the baseline (native execution in local memory) with `RACK.RP`, `Infiniswap`, and `SSD`.

The transaction latencies of `Tpcc`, `Twitter`, and `Wikipedia` from OLTP-Bench are shown in Figure 10. The result demonstrates the feasibility of deploying memory disaggregation in an environment running latency-critical applications. For `OLTPBench.tpcc` and `OLTPBench.wikipedia`, `RACK.RP` shows great tolerance even under an intensive memory disaggregation setup at a local memory limit of 30% w.r.t. RSS. For `OLTPBench.tpcc`, the 99th percentile latencies of `BASELINE`, `RACK.RP`, `Infiniswap`, and `SSD` are 246ms, 257ms, 348ms, and 487ms, respectively. `OLTPBench.Twitter`, which has a relatively higher throughput than the other two workloads, is a difficult benchmark under memory disaggregation. Nevertheless, `Rack.RP` is able to significantly reduce the tail latency compared to `Infiniswap` which shows the benefit of RackMem's virtual memory for latency-critical workloads. In `OLTPBench.Twitter` case, finally, the 99th percentile latency of `BASELINE`, `RACK.RP`, `Infiniswap`, and `SSD` are 5.5ms, 12.4ms, 25.7ms, and 67.3ms, respectively.

5.5 Network Bandwidth Analysis

A common concern of distributed memory is network bandwidth utilization. Frequent accesses to remote memory can potentially saturate a node's network bandwidth and cause prolonged pagefault handling latencies. While saturation can occur at the recipient (local) and the remote memory donor nodes, the network of the local node is more likely to be congested under heavy loads because remote memory is distributed to several donor nodes (Section 4.2).

Figure 9 plots the network bandwidth utilization of RackMem with prefetch and reclaim activated (`RACK.RP`) for all applications from Figure 7. The boxplot visualizes the bandwidth distribution at one-second intervals for each memory configuration. The black and red lines indicate the mean and median values, the boxes represent the 25^{th} and $75th$ percentile, and the bottom/top whiskers show the 5^{th} and 95^{th} percentile of the distribution. The results show an exponential increase in the consumed network bandwidth as the local memory limit is lowered. At the 30% local memory limit, the 95^{th} percentile bandwidth of a single application reaches 1.15 GB/s. With HDR InfiniBand speeds reaching 200 Gb/s (24 GB/s) [39], network bandwidth only gets saturated when 20 parallel applications running at a 30% local memory limit simultaneously reach peak network bandwidth utilization. The FDR InfiniBand setup of our cluster (56 Gb/s or 6.5 GB/s) can support five parallel applications a 30% local memory without suffering from bandwidth saturation.

RackMem has the ability to automatically re-balance local memory to the co-located applications (Section 5.6); as a consequence, it is unlikely that co-located applications all consume peak network bandwidth utilization in a realistic scenario. Indeed, the analysis in Section 2.2 showed that RackMem achieves close to optimal performance at a 50% local memory ratio even without automatic re-balancing.

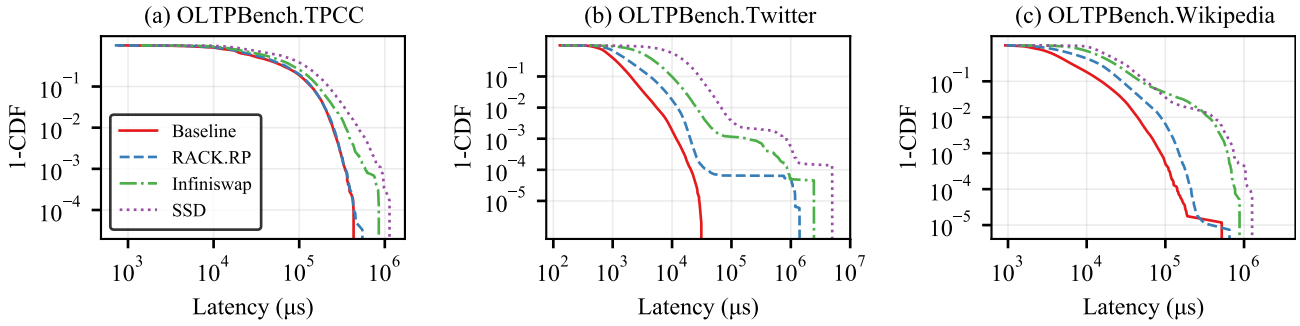

Figure 10: Distribution of transaction latency of OLTP-Bench (log scale, lower is better).

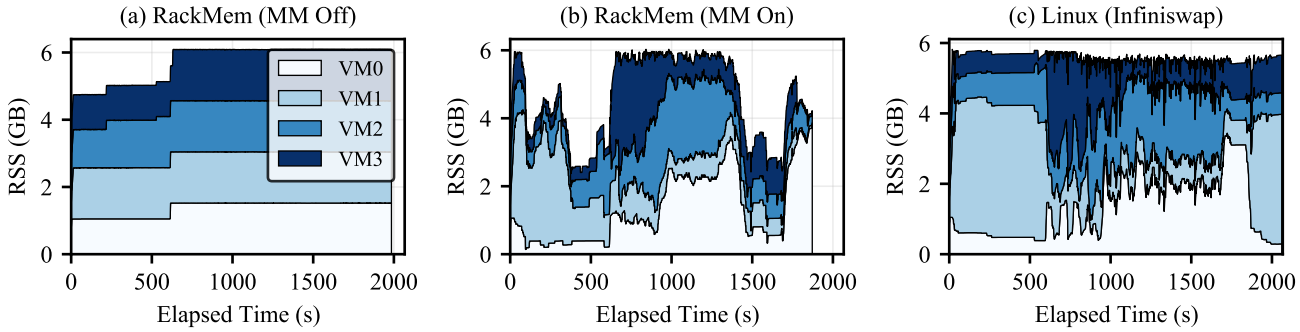

Figure 11: Comparison of dynamic memory balancing of Linux and RackMem userspace memory manager.

5.6 Dynamic Local Memory Partitioning

Multiple applications are often co-located on the same physical machine to improve resource utilization. An important and interesting problem in such a scenario is to allocate the right amount of resources to the different applications. Modern data center applications exhibit a dynamically changing working set which makes it difficult to distribute resources statically. This section evaluates the dynamic memory re-allocator of RackMem (Section 4.3) that redistributes the amount of local memory in proportion to the number of pagefaults generated by the concurrently running applications over a 60-second window.

For the evaluation, four virtual machines (VM) with 4 VCPUs and 8 GB of RAM are co-located. Each VM serially executes 20 random batch processing workloads until completion. The total available local memory is limited to 6 GB, i.e., 1.5 GB per VM with an equal partitioning. This setup is executed with RackMem's static and dynamic partitioning and compared to Infiniswap. Since Infiniswap does support per-application memory limits, the four VMs are executed in a cgroup with 6 GB or RAM. Measured performance metrics are the mean job completion time and the total time to completion defined by the point in time when all VMs have completed their workloads.

Figure 11 plots the results. Figure 11 (a) shows the results for RackMem without a dynamic memory manager. Once allocated memory is never rebalanced. All VMs receive a similar amount of memory; the time to completion is 1'986s. Figure 11 (b) plots the results with RackMem's dynamic memory repartitioning. VMs are assigned more or less memory depending on their number of pagefaults. We observe that all VMs exhibit different phases and that the phases of high memory requirements do not necessarily occur simultaneously. This allows RackMem's dynamic memory partitioner to allocate more memory to VMs when required. VM 3, for example, is assigned up to 3.5 GB at the 600 second mark. With memory re-balancing, the time to completion is 1'873s, a 5.6% reduction compared to the static allocation under RackMem. Finally, Figure 11 (c) shows the result for Infiniswap with a total time to completion of 2'062s. The mean job completion times of RackMem with/without dynamic repartitioning and Infiniswap are 354s, 362s, and 400s, respectively.

5.7 Job Processing on a Real Cluster

The final experiment validates the simulated results of the job queue from Section 2.2 and Figure 2. The cluster from Section 5.1 is setup to processes 100 batch workloads selected randomly from the 12 workloads presented in Section 5.4.1. Each node is configured to utilize only 12.5% of its local memory and request remote memory if the running jobs exceed the local memory limit. Jobs are scheduled with the same conditions as the simulation from Section 2.2. The evaluation measures the mean job turnaround time for three different configurations `RACK.RP`, `Infiniswap`, and `SSD`.

Figure 12 (a) shows the CDF of the job turnaround times. The ability of memory sharing beyond physical limits improves the job processing time for both `RACK.RP` and `Infiniswap`. The mean job turnaround times of `RACK.RP`, `Infiniswap`, and `SSD` are 153.0s, 184.9s, and 260.5s, respectively. Figure 12 (b) shows the aggregated memory utilization of the cluster during the experiment. Both RackMem and

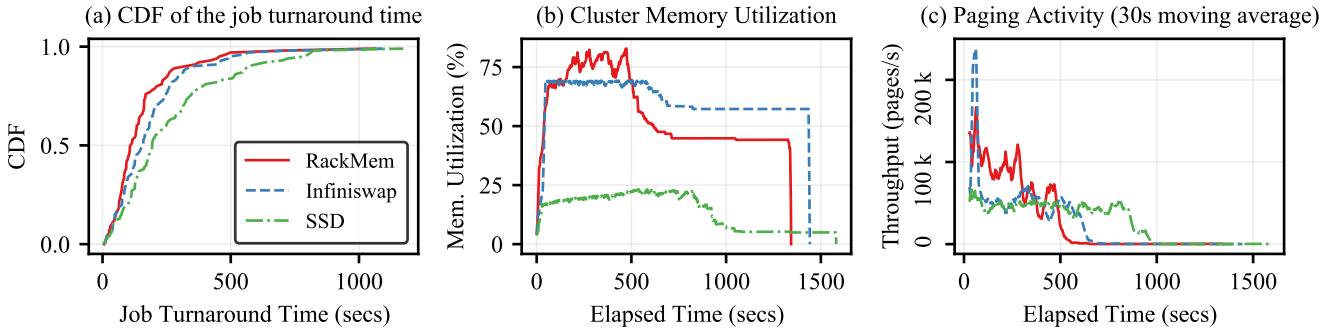

Figure 12: Total turnaround time of batch job processing.

Infiniswap aggressively utilize remote memory and achieve a much higher memory utilization than demand paging to local storage (SSD). Figure 12 (c) plots the aggregated paging throughput of the cluster. The figure plots the 30-second moving average at one second intervals to remove noise and emphasize the changing phases. Except for a short period at the beginning, RackMem consistently handles more remote page requests than Infiniswap or Linux until a steady-state is achieved. Thanks to RackMem's ability to utilize remote memory more aggressively, it achieves higher throughput and shorter job processing times.

6 RELATED WORK

Accessing remote memory over a fast network is not an entirely new idea, and a number of works address the different aspects of remote memory systems.

Hardware support for memory disaggregation. Several works present specialized hardware to transparently access remote memory without the intervention of software pagefault handlers. Lim *et al.* [28, 29] proposed a new architecture that transparently utilized a pool of memory blades with a modified hypervisor. The authors show that simple modifications to the hardware, combined with matching software, can significantly improve the memory utilization in a cluster. The necessity for custom hardware is the main limitation of this work. More recently, Katrinis *et al.* [23] have presented a working prototype of hardware disaggregated memory. While the work represents a promising design for future data centers, it will likely take years until a commercial product is available and integrated in data centers – a time during which software-based disaggregated memory is a viable and attractive solution.

Demand paging to remote memory. Using remote memory as an extension of local memory has a long history. Newhall *et al.* [41] proposed Nswap, a concept and implementation of a remote swap system. The authors showed that paging to remote memory has a performance advantage over swapping to a hard disk thanks to the smaller latency of the network. The use of advanced commercial networking devices such as Infiniband with RDMA has been explored by Liang *et al.* [27]. RDMA and the much faster Infiniband network significantly improve the performance of remote paging and simplify the software implementation. More recently, Gao *et al.* [16] have analyzed the feasibility of resource disaggregation and presented a comprehensive analysis of network requirements. The analysis shows that available high-speed network devices offer sufficient performance to realize memory disaggregation with an efficient software implementation. Gu *et al.* [18] presented Infiniswap, a practical implementation of a distributed remote swap device. Infiniswap uses unused memory in the cluster as a swap space for local memory. Fault tolerance is achieved by simultaneously writing to remote storage and a local disk; if the remote node becomes unavailable, the data is recovered from the disk. Similar to our work, Infiniswap employs the power-of-two-choices algorithm to select a remote node. The work of Koh *et al.* [25] is closest to RackMem. The authors presented a custom swap layer over RDMA implemented in the KVM/QEMU hypervisor. The custom swap layer outperforms Linux's swapping over RDMA thanks to scalable data structures and elastic block management for prefetching. The contribution of RackMem over this work is the design of an efficient remote paging module with well-designed and powerful interfaces. With modifications to only a few lines of code in the KVM/QEMU hypervisor, virtual machines in data centers are able to benefit from RackMem's disaggregated memory and advanced cache manager.

7 CONCLUSION

This paper has presented RackMem, an efficient implementation of disaggregated memory implemented and evaluated with a recent Linux kernel. RackMem exposes distributed memory through a simple device interface. Local memory is used as a cache. RackMem allows dynamic and detailed configuration of its operations at runtime from user-space, rendering it an ideal candidate to implement user-level and application-specific caching policies. Experiments with a wide range of real-world data center applications show the benefits of RackMem compared to the state-of-the-art RDMA paging and local paging to SSDs using Linux's virtual memory demand paging.

ACKNOWLEDGMENTS

This work was supported by the National Research Foundation of Korea (NRF) funded by the Korean government, in part, by grants NRF-2015K1A3A1A14021288, 2016R1A2B4009193, by the BK21 Plus for Pioneers in Innovative Computing (Dept. of Computer Science and Engineering, SNU, grant 21A20151113068), and by the Promising-Pioneering Researcher Program of Seoul National University in 2015. ICT at Seoul National University provided research facilities for this study.

REFERENCES

[1] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2018. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, 775–787. https://www.usenix.org/conference/atc18/presentation/aguilera

[2] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2017. Remote Memory in the Age of Fast Networks. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*. ACM, 121–127. https://doi.org/10.1145/3127479.3131612

[3] Alibaba. 2018. Alibaba Production Cluster Trace Data. https://github.com/alibaba/clusterdata.

[4] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Russell L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. 1991. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications* 5, 3 (1991), 63–73.

[5] Jeff Barr. 2019. EC2 High Memory Update âĂŞ New 18 TB and 24 TB Instance. https://aws.amazon.com/ko/blogs/aws/ec2-high-memory-update-new-18-tb-and-24-tb-instances/.

[6] M. Bielski, Ilias Syrigos, Kostas Katrinis, Dimitris Syrivelis, Andrea Reale, Dimitris Theodoropoulos, Nikolaos Alachiotis, Dionisios Pnevmatikatos, H. E. Pap, George Zervas, Vaibhawa Mishra, A. Saljoghei, A. Rigo, J. F. Zazo, Sergio Lopez-Buedo, Marti Torrents, Ferad Zyulkyarov, M. Enrico, and O. G. de Dios. 2018. dReDBox: Materializing a full-stack rack-scale system prototype of a next-generation disaggregated datacenter. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1093–1098. https://doi.org/10.23919/DATE.2018.8342174

[7] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph.D. Dissertation. Princeton University.

[8] Brad Calder and Bart Sano. 2019. Introducing Compute- and Memory-Optimized VMs for Google Compute Engine. https://cloud.google.com/blog/products/compute/introducing-compute-and-memory-optimized-vms-for-google-compute-engine.

[9] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and QoS-aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, 127–144. https://doi.org/10.1145/2541940.2541941

[10] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.* 7, 4 (Dec. 2013), 277–288. https://doi.org/10.14778/2732240.2732246

[11] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan Milojicic. 2015. Beyond Processor-centric Operating Systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. USENIX Association. https://www.usenix.org/conference/hotos15/workshop-program/presentation/faraboschi

[12] Linux Foundation. 2019. mm, swap: use rbtree for swap extent. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=4efaceb1c5f8136d5fec3f26549d294b8e898bd7.

[13] Linux Foundation. 2020. cgroups(7) - Linux manual page. http://man7.org/linux/man-pages/man7/cgroups.7.html.

[14] Linux Foundation. 2020. Linux kernel documentation. https://www.kernel.org/doc/Documentation/trace/tracepoints.txt.

[15] Linux Foundation. 2020. Null block device driver. https://www.kernel.org/doc/html/latest/block/null_blk.html.

[16] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network Requirements for Resource Disaggregation. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, 249–264. http://dl.acm.org/citation.cfm?id=3026877.3026897

[17] Google. 2011. Google Production Cluster Trace Data. https://github.com/google/cluster-data.

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

[19] Infiniswap 2017. Infiniswap: Efficient Memory Disaggregation with Infiniswap. https://github.com/SymbioticLab/infiniswap.

[20] Intel. 2018. Intel Rack Scale Design Architecture. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/rack-scale-design-architecture-white-paper.pdf.

[21] Adam Jacobs. 2009. The pathologies of big data. *Commun. ACM* 52, 8 (2009), 36–44.

[22] Karthik Kambatla, Giorgos Kollias, Vipin Kumar, and Ananth Grama. 2014. Trends in big data analytics. *J. Parallel and Distrib. Comput.* 74, 7 (2014), 2561 – 2573. https://doi.org/10.1016/j.jpdc.2014.01.003 Special Issue on Perspectives on Parallel and Distributed Processing.

[23] Kostas Katrinis, Dimitris Syrivelis, Dionisios Pnevmatikatos, George Zervas, Dimitris Theodoropoulos, Iordanis Koutsopoulos, K. Hasharoni, Daniel Raho, Christian Pinto, F. Espina, Sergio Lopez-Buedo, Q. Chen, Mario D. Nemirovsky, Damian Roca, H. Klos, and T. Berends. 2016. Rack-scale disaggregated cloud data centers: The dReDBox project vision. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*. 690–695.

[24] Aniraj Kesavan, Robert Ricci, and Ryan Stutsman. 2017. To Copy or Not to Copy: Making In-Memory Databases Fast on Modern NICs. In *Data Management on New Hardware*. Springer International Publishing, 79–94.

[25] Kwangwon Koh, Kangho Kim, Sunghyub Jeon, and Jaehyuk Huh. 2018. Disaggregated Cloud Memory with Elastic Block Management. *IEEE Trans. Comput.* (2018), 1–1. https://doi.org/10.1109/TC.2018.2851565

[26] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. 2019. Software-Defined Far Memory in Warehouse-Scale Computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, New York, NY, USA, 317–330. https://doi.org/10.1145/3297858.3304053

[27] Shuang Liang, Ranjit Noronha, and Dhabaleswar K. Panda. 2005. Swapping to Remote Memory over InfiniBand: An Approach using a High Performance Network Block Device. In *2005 IEEE International Conference on Cluster Computing*. 1–10. https://doi.org/10.1109/CLUSTR.2005.347050

[28] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. 2009. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA âĂŹ09)*. Association for Computing Machinery, New York, NY, USA, 267âĂŞ278. https://doi.org/10.1145/1555754.1555789

[29] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. 2012. System-level implications of disaggregated memory. In *IEEE International Symposium on High-Performance Comp Architecture*. 1–12. https://doi.org/10.1109/HPCA.2012.6168955

[30] H. Litz, M. Thuermer, and U. Bruening. 2010. TCCluster: A Cluster Architecture Utilizing the Processor Host Interface as a Network Interconnect. In *2010 IEEE International Conference on Cluster Computing*. 9–18. https://doi.org/10.1109/CLUSTER.2010.37

[31] Feilong Liu, Lingyan Yin, and Spyros Blanas. 2017. Design and Evaluation of an RDMA-aware Data Shuffling Operator for Parallel Database Systems. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, 48–63. https://doi.org/10.1145/3064176.3064202

[32] Yin Lu, Yong Chen, Yu Zhuang, Jialin Liu, and Rajeev Thakur. 2015. Collective input/output under memory constraints. *International Journal of High Performance Computing Applications* 29, 1 (2015), 21–36. https://doi.org/10.1177/1094342014561696

[33] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: an RDMA-enabled Distributed Persistent Memory File System. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, 773–785. https://www.usenix.org/conference/atc17/technical-sessions/presentation/lu

[34] LWN.net. 2016. Making swapping scalable [LWN.net]. https://lwn.net/Articles/704478/.

[35] LWN.net. 2017. mm, swap: VMA based swap readahead [LWN.net]. https://lwn.net/Articles/716296/.

[36] LWN.net. 2017. The next steps for swap [LWN.net]. https://lwn.net/Articles/717707/.

[37] Hasan Maruf and Mosharaf Chowdhury. 2019. Effectively Prefetching Remote Memory with Leap. (11 2019).

[38] Mellanox. 2016. Mellanox Products: ConnectXÂő-5 Single/Dual-Port Adapter supporting 100Gb/s with VPI. http://www.mellanox.com/page/products_dyn?product_family=258&mtag=connectx_5_vpi_card.

[39] Mellanox. 2020. Introducing 200G HDR InfiniBand Solutions. https://www.mellanox.com/pdf/whitepapers/WP_Introducing_200G_HDR_InfiniBand_Solutions.pdf.

[40] Michael Mitzenmacher. 2001. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems* 12, 10 (Oct 2001), 1094–1104. https://doi.org/10.1109/71.963420

[41] Tia Newhall, Sean Finney, Kuzman Ganchev, and Michael Spiegel. 2003. Nswap: A Network Swapping Module for Linux Clusters. In *Euro-Par 2003 Parallel Processing*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1160–1169.

[42] Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, and Daniel Hagimont. 2018. Welcome to Zombieland: Practical and Energy-efficient Memory Disaggregation in a Datacenter. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. ACM, Article 16, 12 pages. https://doi.org/10.1145/3190508.3190537

[43] pmem.io. 2020. Persistent Memory Programming. http://pmem.io/.

[44] Stephen M. Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K. Ousterhout. 2011. It's Time for Low Latency. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems (HotOS'13)*. USENIX Association, 11–11. http://dl.acm.org/citation.cfm?id=1991596.1991611

[45] Iman Sadooghi, Jesus Hernandez Martin, Tonglin Li, Kevin Brandstatter, Ketan Maheshwari, Tiago Pais Pitta De Lacerda Ruivo, Gabriele Garzoglio, Steven Timm, Yong Zhao, and Ioan Raicu. 2017. Understanding the Performance and Potential of Cloud Computing for Scientific Applications. *IEEE Transactions on Cloud Computing* 5, 2 (2017), 358–371. https://doi.org/10.1109/TCC.2015.2404821

[46] Corey Sanders. 2018. Why you should bet on Azure for your infrastructure needs, today and in the future. https://azure.microsoft.com/en-us/blog/why-you-should-bet-on-azure-for-your-infrastructure-needs-today-and-in-the-future/.

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

[48] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. 2017. Distributed Shared Persistent Memory. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*. ACM, 323–337. https://doi.org/10.1145/3127479.3128610

[49] Avinash Sodani. 2011. Race to exascale: Challenges and opportunities. *MICRO 2011 Keynote* (2011).

[50] Petter Svärd, Benoit Hudzia, Johan Tordsson, and Erik Elmroth. 2014. Hecatonchire: Towards Multi-host Virtual Machines by Server Disaggregation. In *Euro-Par 2014: Parallel Processing Workshops*. Springer International Publishing, 519–529.

[51] Top500.org. 2020. TOP500 Supercomputer Sites. https://top500.org/.

[52] Shin-Yeh Tsai and Yiying Zhang. 2017. LITE Kernel RDMA Support for Datacenter Applications. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, 306–324. https://doi.org/10.1145/3132747.3132762

[53] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. USENIX Association, 10–10. http://dl.acm.org/citation.cfm?id=1863103.1863113

[54] Darko Zivanovic, Milan Pavlovic, Milan Radulovic, Paul M. Carpenter, Petar Radojkovic, Eduard Ayguade, Hyunsung Shin, Jongpil Son, and Sally A. McKee. 2017. Main memory in HPC: Do we need more or could we live with less? *ACM Transactions on Architecture and Code Optimization* 14, 1 (2017), 1–26. https://doi.org/10.1145/3023362