Ph.D. DISSERTATION

Remote Memory for Virtualized Environments

가상화 환경을 위한 원격 메모리

AUGUST 2021

DEPARTMENT OF ELECTRICAL ENGINEERING &
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Changyeon Jo

Remote Memory for Virtualized Environments

가상화 환경을 위한 원격 메모리

지도교수 Bernhard Egger

이 논문을 공학박사 학위논문으로 제출함

2021 년 4 월

서울대학교 대학원

전기·컴퓨터공학부

조 창 연

조 창 연의 공학박사 학위논문을 인준함

2021 년 7 월

위 원 장	이 재 진
부위원장	Bernhard Egger
위 원	엄 현 상
위 원	Thomas Gross
위 원	Patrick Stuedi

Abstract

The raising importance of big data and artificial intelligence (AI) has led to an unprecedented shift in moving local computation into the cloud. One of the key drivers behind this transformation was the exploding cost of owning and maintaining large computing systems powerful enough to process these new workloads. Customers experience a reduced cost by renting only the required resources and only when needed, while data center operators benefit from efficiency at scale.

A key factor in operating a profitable data center is a high overall utilization of its resources. Due to the scale of modern data centers, small improvements in efficiency translate to significant savings in the total cost of ownership (TCO).

There are many important elements that constitute an efficient data center such as its location, architecture, cooling system, or the employed hardware. In this thesis, we focus on software-related aspects, namely the utilization of computational and memory resources. Reports from data centers operated by Alibaba and Google show that the overall resource utilization has stagnated at a level of around 50 to 60 percent over the past decade [29,64]. This low average utilization is mostly attributable to peak demand-driven resource allocation despite the high variability of modern workloads in their resource usage. In other words, data centers today lack an efficient way to put idle resources that are reserved but not used to work.

In this dissertation we present RackMem, a software-based solution to address the problem of low resource utilization through two main contributions. First, we introduce a disaggregated memory system tailored for virtual environments. We observe that virtual machines can use remote memory without noticeable performance degradation under moderate memory pressure on modern networking infrastructure. We implement a specialized remote paging system for QEMU/KVM that reduces the remote paging tail-latency by 98.2% in comparison to the state of the art. A job processing simulation at rack-scale shows that the total makespan can be reduced by 40.9% under our memory system.

While seamless disaggregated memory helps to balance memory usage across nodes, individual nodes can still suffer overloaded resources if co-located workloads exhibit high resource usage at the same time. In a second contribution, we present a novel live migration technique for machines running on top of our remote paging system. Under this instant live migration technique, entire virtual machines can be migrated in as little as 100 milliseconds. An evaluation with in-memory key-value database workloads shows that the presented migration technique improves the state of the art by a wide margin in all key performance metrics.

The presented software-based solutions lay the technical foundations that allow data center operators to significantly improve the utilization of their computational and memory resources. As future work, we propose new job schedulers and load balancers to make full use of these new technical foundations.

Keywords: virtualization, live migration, remote memory, RDMA
Student Number: 2012-20869

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The volume of data centers has been increasing rapidly in the last decade with the rising interest of artificial intelligence (AI), machine learning (ML), and big data analytics. These workloads require more and more resources (i.e., computation, memory, and storage), and we expect this trend will continue or even increase in the upcoming years.

Since the sheer volume of modern workloads is very large, the role of software runtime in datacenters is increasingly important to better utilize the limited resources in data centers. Especially, many modern workloads are running in the virtualized environments (i.e., 70% percent, according to an industry report [4]), which implies improving virtualized environments is the key to the above problem.

Virtualization gives many advantages for better utilizing system resources by consolidating more workloads on the same machine and deal with the workload variability with live migration (i.e., moving the workloads on a heated server to other machines). It has been widely adopted in production environments [58] and providing the foundation of modern clouds serving billions of users.

On the other hand, virtualization is still experiencing efficiency problems. We

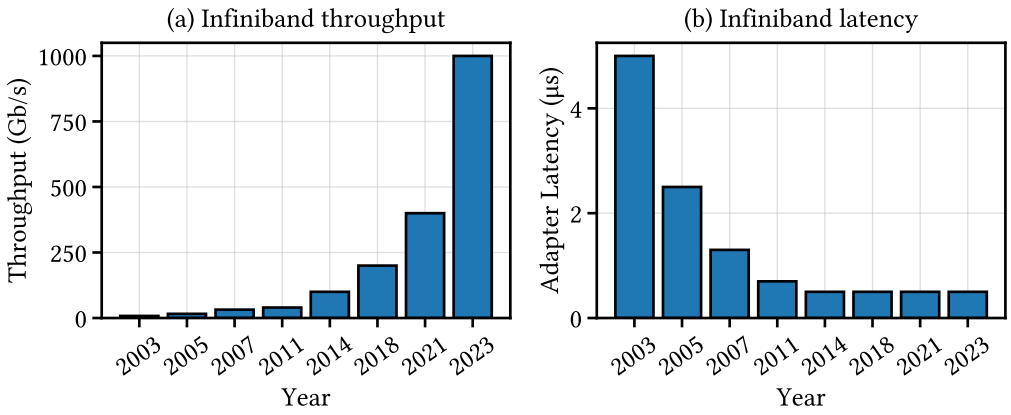


Figure 1.1 Infiniband adapter performance.

have seen a significant portion of unutilized resources in data centers; The recent publications from Google and Alibaba show the average utilization of CPU and memory in their clusters are stagnated at 60% and 40% levels separately in 2019 [29, 64]. It is quite impressive improvement compared to 40% utilization of the 2011 Google traces [57]; however, we still have a large room for improvement at the top.

There are two major reasons that fundamentally limit the efficiency of virtualized environments. First, a machine cannot utilize resources beyond the physical machine's boundary. Especially memory is difficult to share between machines due to the lack of efficient sharing mechanism and a significant gap between the local and remote access latency.

Second, modern workloads are highly volatile in their resource usage, making it difficult to co-locate multiple workloads on the same machine to improve resource utilization. Aggressive memory overcommit increases the chance of SLOs (Service Level Objectives) violation, and it forces to keep a non-negligible amount of resources in each server idle. Virtual machine (VM) live migration is a promising solution for handling the resource variability since it allows a VM can be moved between servers without noticeable downtime. However, VM live migration requires sending all states (i.e., the whole memory of a VM) to the destination server, and the cost is prohibitively high with large VM instances, which are common these days.

Memory disaggregation has been proposed recently as a promising solution to

address the problems mentioned above. The major motivation of memory disaggregation lies in the remarkable performance of modern commodity networks. Commodity networking technologies provide a sub-microsecond remote memory access latency with more than 100gbit/s bandwith [49].

Infiniband standard also doubling the performance by every a few years. Figure 1.1 shows the Infiniband standard improvement over years in throughput and latency. A 4-lane Infiniband adapter will be expected to provide 1,000 Gb/s of throughput with sub-microsecond latency in 2023, rapidly blurring the boundary between local and remote memory. By building a shared memory pool accessible through the fast network, memory is easily shared between the machines in the same cluster. It helps to increase memory utilization by making idle memory is utilized by remote machines and has been proved its usability in many articles in the recent days [12, 14, 27, 28, 42, 44, 45, 59].

However, the software-based memory disaggregation still experiences suboptimal performance due to the poorly optimized software stack, and lack of mechanism for seamless and instant VM live migration in virtualized environments. In our thorough evaluation, the state-of-the-art solutions [14, 27, 28] suffer from long tail latencies (two orders of magnitude slower software overhead than the actual I/O at the tail) under intensive remote paging scenarios. Also, despite the benefit of accessing any data in any node with memory disaggregation, VM live migration still suffers from long latency to complete due to the entire memory copy from the source to the destination host. Furthermore, memory disaggregation unexpectedly makes the vanilla VM live migration slower, since it does not properly consider the context of the memory disaggregation.

1.1 Contributions of the Dissertation

In this dissertation, we make the following contributions:

- We present RackMem, a software-based memory disaggregation for virtualized environments that provide ~10 microseconds of remote memory access latency at the 90th percentile. RackMem achieves the significant latency improvement by optimizing the three layers coposing the remote paging: the virtual memory

(Chapter 4), the backend distrubted storage (Chapter 5), and the networking library for remote node communitation (Chapter 6).

- We present instant and seamless virtual machine migration that provides a small constant latency (around 100ms) for VM live migration and minimize performance degradation during VM live migration (Chapter 7).
- We evaluate the proposed techniques with a extensive set of real applications (Chapter 8 and Chapter 9)

By running virtual machines on RackMem, data center operators have two benefits. First, RackMem enables transparent memory sharing for nodes in the same cluster. RackMem effectively hides the remote memory access latency by utilizing the local memory as cache with a number of optimization techniques. Under moderate remote paging scenarios, applications show close to native performance with RackMem. Applications also show high tolerance in performance even under the intensive remote paging scenarios. It provides a higher degree of elasticity for utilizing memory in the cluster and helps to improve the overall memory utilization.

Second, the instant VM live migration quickly (~100ms) resolves hotspot in the cluster. It is useful even if RackMem handles a hotspot by utilizing idle memory in other clusters. By moving a VM to another server, we can utilize more computation resources on the destination host and reduce the remote paging by directly accessing the destination server's local memory. Instant VM live migration is also attractive for the maintenance scenario that virtual machines have to be migrated to safe nodes when VMs experience performance anomaly due to malfunctioning hardware.

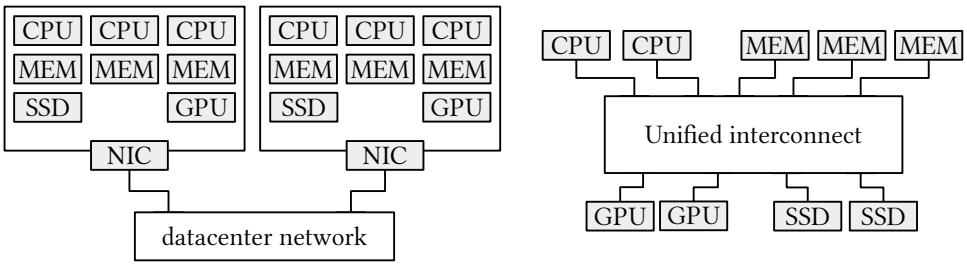
Chapter 2

Background

In this chapter, we provide necessary background for the rest of the dissertation. First, we explain the memory disaggregation, a technique that building a shared memory pool and utilize it through high-performance interconnect technologies. Second, we present the background of remote direct memory access (RDMA). RDMA provides an efficient mechanism for remote memory access bypassing the kernel in I/O operations and completely avoid the interaction with the remote server. Third, the live migration of virtual machines which is a core mechanism for the seamless operation of datacenters running virtualized workloads.

2.1 Resource Disaggregation

Traditionally a single task is allowed to utilize only the local resources (i.e., resources connected on the same main board) which we call *"server centric"* architecture (Figure 2.1 (a)). In server centric architecture, a job scheduler maintains the state of available resources in each server, and place a task only if the node has enough resources to serve the task's resource requirements. This architecture has a serious disadvantage. When each node in the cluster only partially meets the resource requirements,



(a) node centric (b) resource centric

Figure 2.1 Resource disaggregation

the request will fail even though the cluster has enough aggregated resources to serve the request. Users prone to request more resources than they actually use, so resource overcommit could alleviate the situation. However, this approach has a potential risk of resource depletion since we cannot know future resources usage of the co-located tasks. Wrong resource overcommit would result in severe slowdown of the tasks or unexpected termination of tasks in the worst case.

The main reason of deploying the server centric architecture as common practice is the performance gap between local and remote access. If we consider remote memory access latency, it was a few orders of magnitude slower than the local memory access. To make it matter worse, utilizing remote resources usually requires a significant modification in applications. Cache coherent distributed memory cannot solve the problem, since the high cost of software based cache coherency will quickly eliminate the benefit of utilizing remote resources.

However, the relentless effort for improving the networking hardware performance, the idea of utilizing remote resouces has been reviving in the recent days. The performance of modern networking hardware is closing to the speed-of-light, and blurs the boundary between local and remote accesses. The latest network interface card (NIC) from NVIDA, ConnectX-7, advertises its sub-microsecond latency and 400Gb/s throughput on PCIe 5.0 [3]. The advances in the networking hardware result in *"resource centric"* architecture (Figure 2.1 (b)) which is a promising solution for the resource fragmentation problem. In resource centric architecture, individual resource pools (e.g., CPU, memory, storage, accelerators) are connected through fast interconnect technology to provide a single cloud computer image. Resource centric

architecture provides high elasiticy in resource management. Resource fragmentation does not exists anymore, and dynamic resource scaling is easy since the all resources are available through the unified interconnect.

Ideal resource disaggregation will come with a special hardware supports [1, 2, 9], however, there are still many challenges until we get commodity products at the market. Inventing a new hardware costs a lot and there is a risk of failure that the new hardware is not adopted by the standard in the market. Or, they may not be realized, we have seen a number of abandoned industry projects [6, 7] in the recent years.

Software-based solutions still attractive and provides the similar benefits with off-the-shelf NICs. Compared to the hardware-based approaches, it even gives more benefits in terms of applicability and configurability. Legacy systems can benefit from the software-based solution, and it is also possible to implement various resource management policies in software.

In addition to that, current operating systems are already prepared for the memory disaggregation. The *virtual memory* provides the core mechanism to implement the illusion of locally available disaggregated memory with hardware MMU. A number of attempts [14, 27, 28, 48] have shown that simple modification in the swap subsystem in Linux allows running unmodified applications on disaggregated memory and improved the cluster wide memory utilization.

In the rest of this dissertation, we focus on the similar approach for memory disaggregation to better apply it to the virtualized environments.

2.2 Transparent Remote Paging

There are many interfaces for utilizing remote memory. For example, high-performance computing applications explicitly utilize remote memory with specialized interfaces such as MPI and RDMA [41]. Or a key-value store interface [23, 38, 40], file system interface [12, 47, 62], or distributed shared memory [15, 51, 60] can be deployed as a type of higher-level abstraction. In this paper, we target the extreme case of the interface that does not require modification of an application. We call this approach *transparent remote paging*. The transparent remote paging heavily relies on virtual memory. By running an application on virtual memory, we can direct an access request to remote

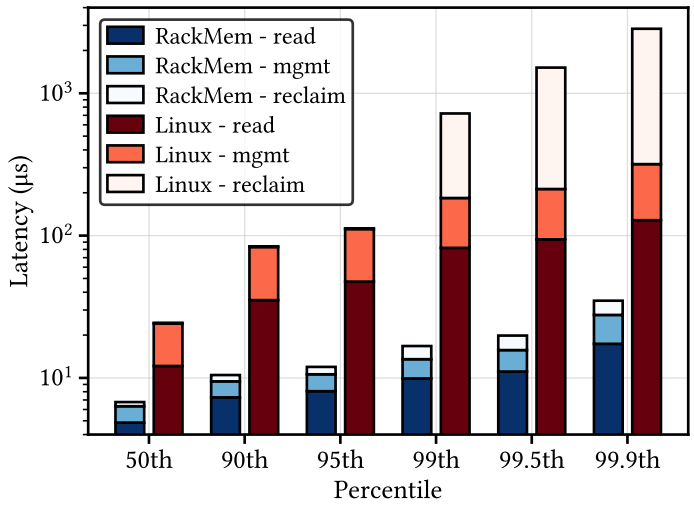


Figure 2.2 Breakdown alanysis of the pagefault handling latency.

memory by using the pagefault handling mechanism.

Transparent remote paging cannot utilize the full potential of the performance benefit of using remote memory. The application often performs unnecessary I/O for small data, and difficult to apply optimization that utilizing the memory access pattern of applications. However, this approach can be utilized without any modification of applications, which is appropriate for production environments that modifying user applications is prohibited.

The common approach for realizing the transparent remote paging is using the swap subsystem that is available in most operating systems. We have seen a recent remote paging system that are built on top of Linux swap subsystem with remote memory storage [14, 28, 48, 53]. Enable remote memory paging in Linux is simple. A block device using remote memory as a storage backend is registered as a backend of swap subsystem. Then, Linux swaps out data to remote memory under memory pressure for unmodified applications.

However, the current implementation of Linux virtual memory's poor performance prevents deploying such a remote paging system for production environments. Figure 2.2 plots the latency of Linux' pagefault handler (kernel version 5.3.9) with In-

finiswap [28], a state-of-the-art transparent remote paging system built on the Linux swap subsystem, and compared its performance to ours 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[1]. The bars in Figure 2.2 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 severe bottleneck in the software that hampers exploiting the full potential of fast disaggregated memory.

2.3 Remote Direct Memory Access (RDMA)

In Section 2.1 we have introduced a high-performance commodity NIC providing ultra-low latency with high bandwidth [3]. The major performance benefit of such hardware comes from the "end host architecture," which allows direct access of the hardware to userspace applications.

With direct access, userspace application enjoys the performance benefits provided by the hardwares such as:

- Kernel bypass eliminates the context switching overhead between the userspace and the kernel for sending I/O requests to the hardware. It significantly reduces the inter-node communication latency.
- Remote direct memory access, allowing direct remote memory without involving the remote CPU in the I/O processing. It minimizes the computation over-

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

head for I/O processing and improves the networking performance.

However, the benefits do not come for free. There are a few challenges for achieving the high-performance:

- Writing an optimized application using the direct access hardware is significantly more complex than traditional applications such as using POSIX socket. For example, writing an application in `ibverbs` [10] (a standard library for writing RDMA applications) requires significantly more lines of code for the same functionality.
- Commodity RDMA devices have scarce resource on the device. Without careful considerations on the resource management, it limits the scalability of RDMA applications.

In this dissertation, we present a system that actively utilizes RDMA for minimizing latency and throughput. We present our approach for achieving the performance benefits and the techniques for overcoming the challenges in Chapter 4, Chapter 5, and Chapter 6.

2.4 Live Migration of Virtual Machines

VM live migration is a technique that is moving a running VM to another server without a (mostly) noticeable downtime. VM live migration is necessary for data centers' operation by enabling seamless maintenance of hardware, software updates, and load balancing.

VM migration requires entire states (e.g., CPU states, memory, VM metadata) copy of the target VM to the destination host; otherwise, the VM cannot continue the execution after the migration. Since the memory state can be several terabytes, it takes the most of the overhead of VM migration.

The simplest way to migrate a VM is *stop-and-copy* that stops the target VM, transfers the entire states to the destination, and resumes the VM. This technique, however, suffers from long downtime since the VM cannot respond until the huge amount of data transfer is finished. Due to the above reason, stop-and-copy cannot deliver the *live* for VM migration.

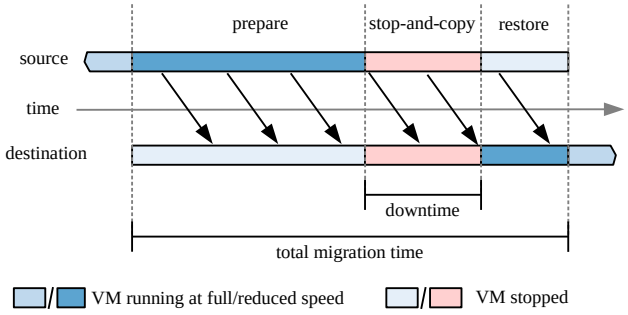


Figure 2.3 VM live migration overview.

Two main approaches bring *live* to VM migration.

Pre-copy [19, 52] iteratively transfers the modified memory of a VM. The first iteration sends the entire memory to the destination node. If data is transferred faster than modified, the iterative process will converge. Once the modified data falls below a given threshold, the VM is stopped, the remaining modified data copied, and then the VM resumes on the destination node.

Post-copy [11,31] sends only the CPU state and immediately resumes the VM on the destination node. Since the migrated VM starts without its memory, any access to yet unrestored pages generate a pagefault. Typically a background process assists the pagefault handler in bringing pages to the destination node. Unlike pre-copy, the entire memory is transferred exactly once, however, VMs often experience a severe performance degradation after migration.

Hybrids between pre- and post-copy have been proposed [61]. Other optimizations employ data compression, CPU throttling, or parallel fetching of data stored on disk [33–35, 46, 63].

Figure 2.3 shows the three phases of VM live migration:

***Prepare*:** VM is running on the source host. Send the VM states to the destination.

***Stop-and-copy*:** VM is not running in this phase. Source side sends remaining VM states to the destination to synchronize the states between two hosts.

***Restore*:** VM is running on the destination host. The destination side fetches the remaining data from the source host to complete the migration.

Stop-and-copy is necessary for all techniques. *Prepare* and *Restore* phases are optional depends on techniques. Only one of them is enabled in some techniques (*pre-copy*, *post-copy*), or both of them are enabled (*post-copy with pre-copy*). Both of them can be disabled when the VM is running on disaggregated memory.

A VM live migration technique is evaluated for the following criteria:

- ***Total Time*:** the period of VM live migration between the beginning and the completion.
- ***Downtime*:** the period of *stop-and-copy* phase.
- ***Transferred Data*:** the total transferred data from the source to the destination to migrate the VM.
- ***Performance*:** the performance degradation of workloads in the target VM during the VM live migration.
- ***Resource Usage*:** the additional resource usages during the live migration (e.g. CPU, memory).

We use the performance metrics for the evaluation of VM live migration techniques in Chapter 9.

Chapter 3

RackMem Overview

Figure 3.1 shows a high-level overview of RackMem. RackMem is a set of components that work cooperatively to provide low-latency remote paging and instant live migration for virtual machines. In this chapter, we give a high-level overview of the core components in RackMem.

3.1 RackMem Virtual Memory

RackMem virtual memory (RackVM) is the user-facing layer of RackMem. Userspace applications can request virtual address spaces to RackVM, and RackVM provides transparent remote paging for the address space. Userspace applications can perform native memory instructions on the address space without worrying about remote paging and resource management.

We design RackVM to minimize the pagefault handling latency and to maximize the paging throughput. To optimize the paging performance, RackVM deploys the pro-active page reclamation, tailored pre-fetching mechanism, and an efficient victim page selection algorithm. We will discuss the design and implementation detail in Chapter 4.

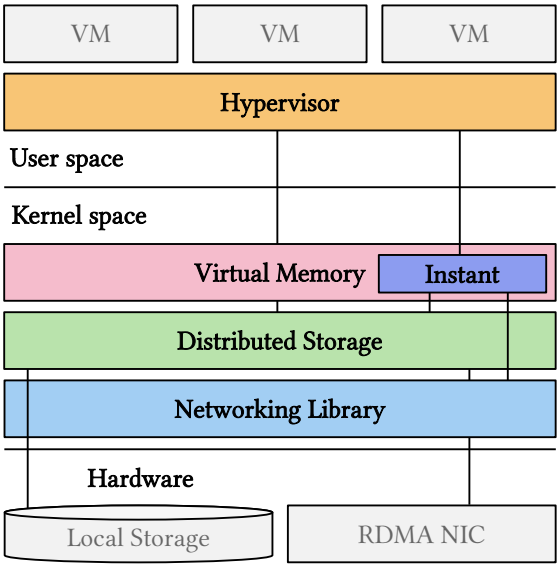


Figure 3.1 An overview and the main components in RackMem.

We also provide a hypervisor that is modified to allocate VM's memory from RackVM. Using virtualization as the main abstraction of the remote memory gives two significant advantages. First, by running applications in VM, unmodified applications can benefit from the transparent remote paging, which makes the proposed system significantly more practical than requiring modification. Second, it also enables a seamless transition from the traditional architecture to the disaggregated architecture. Datacenter administrators can deploy the proposed system without disrupting the existing environments running virtual machines.

3.2 RackMem Distributed Virtual Storage

RackMem distributed virtual storage (RackDVS) provides a byte-addressable storage abstraction of remote memory. RackVM uses RackDVS to allocate backend storage for the managed virtual address space and uses the space to store local pages under memory pressure.

RackDVS provides the three key features that are necessary to support the target virtual memory.

High-performance. RackDVS utilizes the high-performance remote memory (accessed with RDMA) as the main storage. To provide the best performance, RackDVS minimizes the control operation overhead in the data operations; most I/O requests are processed quickly and directly on the hardware.

Reliability. The remote memory can fail anytime; providing reliability is an important goal in RackDVS. RackDVS provides reliability by making a copy of remote page in the local storage with write I/O duplication. We co-design RackVM and RackDVS to hide the I/O duplication overhead in the background.

Multi-storage support. RackDVS deploys a simple design to enable utilizing multiple available storage. This design gives more chances for optimizing the performance per dollar by building a proper hierarchy of storage.

We present the design and the implementation detail of RackDVS in Chapter 5.

3.3 RackMem Networking

RackMem Networking (RackNet) is a kernel space RDMA and RPC library. RackDVS heavily uses RackNet to implement remote memory storage backend, and RackVM also uses it to implement instant VM live migration (Figure 3.1). Usage of RackNet is not limited to RackMem; any other kernel space distributed service can be implemented on top of RackNet. In overall RackNet provides the following core benefits:

Easy to use. The easy programming interface enables quick implementation of distributed services using RDMA. Compared to using `ibverbs` directly, with RackNet, the same feature can be implemented in much shorter lines of code.

Efficient on-device resource utilization. The modern RDMA NIC has known for limited scalability due to the limited on-device resources [39] (e.g., virtual to physical address translation cache). A popular approach for minimizing the on-device resource usage is making multiple applications share the same connection [65]. This approach has known for great scalability than the approach creating a separate connection for each RDMA application. RackNet enables connection sharing by mediating applications in the central space.

Resource efficient kernel-to-kernel RPC. RackNet implements a stateless kernel-

to-kernel RPC that is providing around 10 microseconds latency for a dummy RPC with two-side verbs and interrupt-based completion handling. An RPC client can call a remote function without binding it to the server; this approach does not require additional space to store the state in the memory. RackNet has no busy polling thread to handle the incoming RPC requests. The interrupt-based RPC handling saves a significant amount of computation resources than using a dedicated polling thread.

We present the design and the implementation detail of RackNet in Chapter 6.

3.4 Instant VM Live Migration

The last core feature of RackMem is *instant VM live migration*. We implement the instant VM live migration by extending RackVM to support address space migration to another node and modify the hypervisor to exploit the feature to implement instant VM live migration. (Figure 3.1)

Unlike the traditional VM live migration techniques that require an entire memory copy of VM to the destination, the instant VM live migration only requires transferring a small amount of metadata to the destination describing the page location and the core data structures. The instant VM live migration has the two main performance benefits as follows:

Instant. The re-location of the execution context of a VM is almost instant. Our technique only requires transferring a small amount of data to complete a VM migration; we finish a migration mostly within 100ms.

Seamless. The migrated VM may experience performance degradation for a short period to fetch missing working set pages from the source machine. Our technique quickly recovers from the performance degradation compared to the state-of-the-art implementation [31, 56] by utilizing the significantly faster pagefault handler in RackVM.

We explain the our design and implementation of the instant VM live migration in Chapter 7.

Chapter 4

RackMem Virtual Memory

RackMem virtual memory provides a software-based abstraction for remote memory. Hardware-based transparent remote memory access will be ideal; however, it is not (yet) supported by commodity hardware and is expected to take long years until deployed as a common option in datacenters. RackMem is a software-based approach of remote memory to run applications without a significant performance loss, even for intensive remote paging scenarios. RackMem will be a great software-based solution until hardware-based solutions are realized in the future.

The main design goal of RackVM, while providing reasonable usability, is providing the best possible paging performance comparable to the local memory. Remote memory access is still an order of magnitude slower (600ns) than local memory access (60ns). Without an elaborated design, the remote memory will perform poorly and quickly lose its advantage of utilizing idle resources due to the significant performance degradation. Andres et al. [43] have shown small improvements in memory utilization translate to millions of dollars at a large scale. Hence RackMem's main goal is improving virtual memory performance to exploit the fast storage backends fully (e.g., RDMA and NVMe SSD).

Figure 4.1 shows the main components in RackVM and the counterparts in Linux.

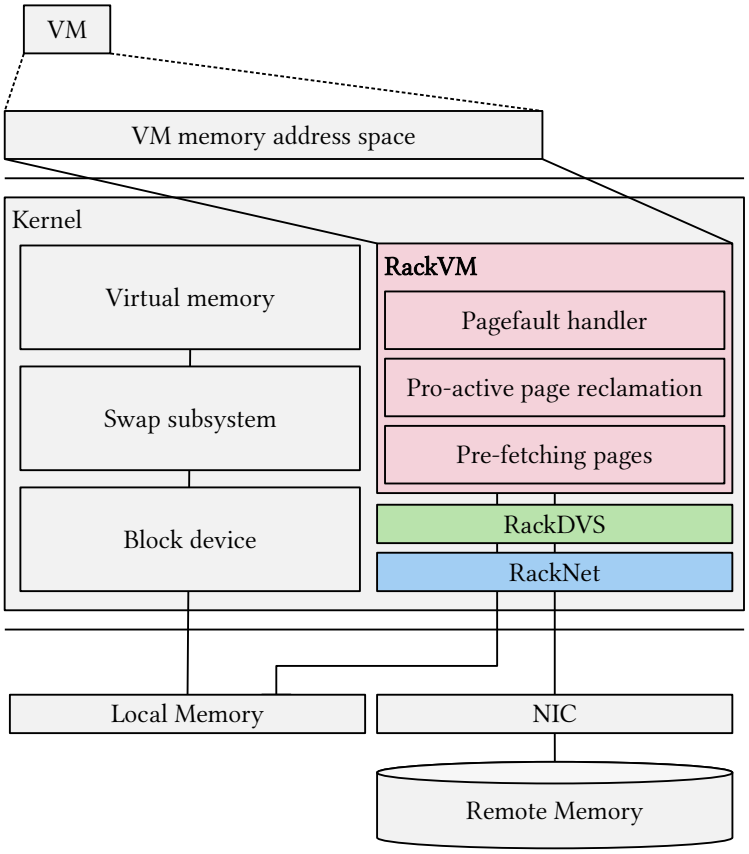


Figure 4.1 The main components in RackVM and the Linux counter parts.

RackVM roughly includes the same functionalities provided by "virtual memory" and "swap subsystem" in Linux. RackVM automatically moves local pages to the backing storage under memory pressure, and minimizes the pagefault handling overheads with the two main optimizations *pro-active page reclamation* and *pre-fetching pages*.

The "pro-active page reclamation" reduces the pagefault handling *latency* by avoiding page-reclamation in the critical path. The "Pre-fetching pages" reduces *the number of pagefaults* by pre-fetching pages if there is a clear sequential memory access pattern.

In the rest of this chapter, we present the design and the implementation detail of RackVM.

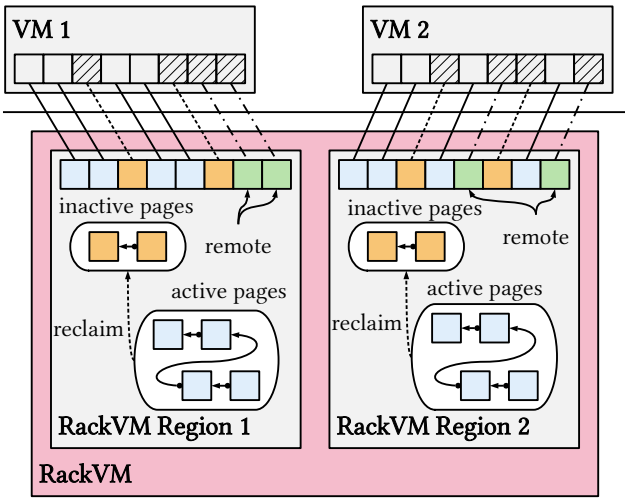


Figure 4.2 Overview of RackVM region.

4.1 Design Considerations for Achieving Low-latency

Figure 4.1 shows an overview of RackVM region. A region is mapped to an entire address space of a VM (or linear virtual address space). To avoid the bottleneck and improve scalability, we design RackVM to maintain separate data structures for each virtual address space.

Virtual memory management. In RackMem, all memory requests are directed to RackVM. RackVM creates a memory *region* associated with the request on the memory allocation request, initializes necessary data structures, and creates a descriptor for the region. RackVM transparently manages the virtual address ranges with the help of hardware page faults. RackVM installs its custom page fault handler for each region to directly handle the page faults and bypasses the default OS handler. Since our abstraction is the virtual memory, user-level applications directly execute memory instructions on the virtual address range.

RackVM manages the region at the internal page granularity and delays the actual memory allocation until the first access. When an application accesses an untouched address, RackVM allocates a new page from the local memory and maps it to the applications' virtual memory space. RackVM does not swap out a page until it hits

```
def rackmem_vm_fault(struct vm_fault *vmf)
{
    /* Step 1: obtain the RackMem region pointer */
    rr = (struct rack_region *) vmf->vma->private_data;
    /* Step 2: lock the page */
    r_page = &rr->pages[index];
    lock(&r_page->lock);
    /* Step 3: Is the pagefault on the prefetched pages? */
    if (r_page->flags & INACTIVE)) { /* yes */
        /* remap the page and insert it to the tail of the active list */
        rack_page_remap(rr, r_page, address);
    } else { /* prefetch miss */
      /*
       * Step 4: Get a free page.
       * The overhead of calling this function is highly dependent on
       * whether it is taking the slowpath or the fastpath.
       */
      rack_get_page(rr, &r_page->buf);
      /* Step 5: restore the page from the backing storage (RackDVS) */
      rack_page_restore(rr, r_page);
    }
    /* Step 6: unlock the page */
    unlock(&r_page->lock);
    /* Step 7: (optional) do proactive page reclamation */
    rack_request_reclamation(rr, nr_pages);
    /* Step 8: (optional) do prefetch */
    rack_prefetch_pages(rr, address, window_size, cache_hit);
}
```

Figure 4.3 Code of the RackMem pagefault handler.

the local memory limit threshold. To handle a page request beyond the local memory limit, RackVM selects a victim page and swap out the page by directing the write request to RackDVS layer. If a swapped-out page is requested later, RackVM restores the page by reading data from RackDVS layer. The local memory limit is configured for each region and dynamically modifiable at any time through the userspace interface.

4.2 Pagefault handling

Quick page fault handling is important in any demand paging system. However, it is even more important when it is running with high-performance storage backends such as remote memory with RDMA and NVMe SSD. Especially, RDMA provides

single-digit microseconds for remote memory access, which means the software overhead of a page fault handler takes a significant portion of the total paging latency.

RackVM deploys a number of techniques to minimize the page fault handling latency while maximizing the throughput and reduce the number of page faults.

First, RackVM handles concurrent page faults on multiples threads. Pagefaults on different pages do not block each other in RackVM. Second, RackVM actively performs prefetch to keep the local memory with pages that will likely be accessed in the near more future. Third, RackVM manages per region free and active lists to avoid contention and tries to keep a small number of pages in the free lists to respond to the paging request immediately. RackVM also reorders the pages in the lists to hold weak LRU properties.

Figure 4.3 shows the annotated code of RackMem's fast page fault handler. We describe the overall page fault handling routine as follows:

1. Obtain the page pointer in the affected region.
2. Lock the page.
3. Lookup the page in the local cache. If found (eg., prefetch hit), map the page into virtual address space and goto step 6.
4. Find a free page in the cache; if no such page exists, select and evict a victim page.
5. If accessed page has been paged out, restore its original data.
6. Unlock the 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 pagefaults.

4.2.1 Fast-path and slow-path in the pagefault handler

Figure 4.3 shows a simplified visualization of the two main paths in the pagefault handler: the *fastpath* and the *slowpath*; Step 3 is omitted for the simplicity. RackVM has been designed to minimize the number of taking the slowpath. We explain the

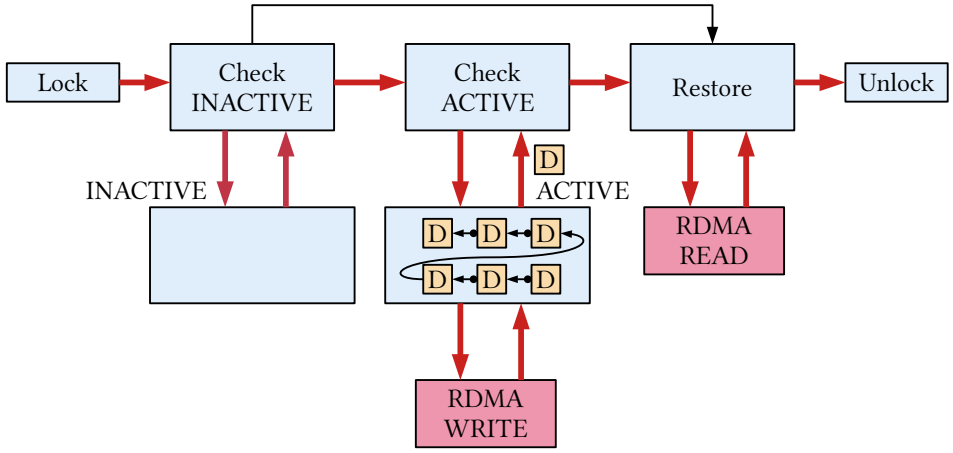


Figure 4.4 The slowpath in the RackMem pagefault handler.

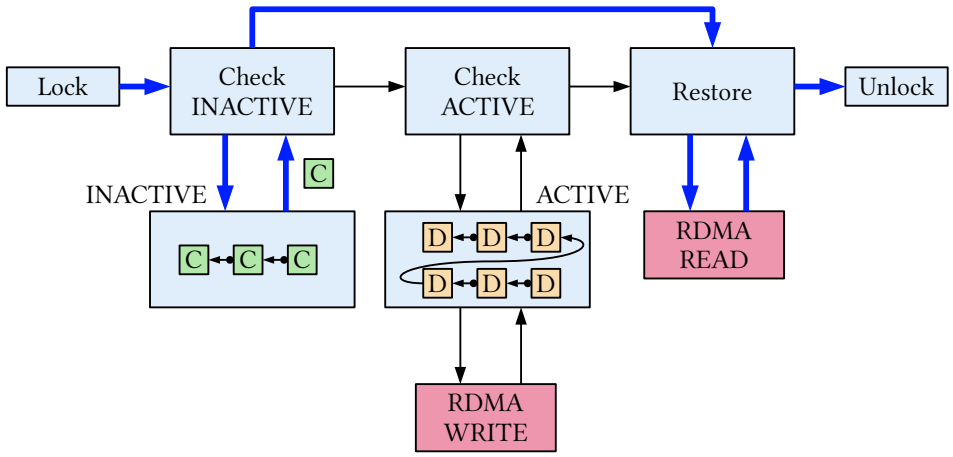


Figure 4.5 the fastpath in the rackmem pagefault handler.

flow of each path in detail as follows.

Slowpath in pagefault handler. Figure 4.4 shows the slowpath in RackVM pagefault handler. In the slowpath, the pagefault handler first checks the `INACTIVE` pool but failed to get a readily available page from the pool. In this case, an `ACTIVE`

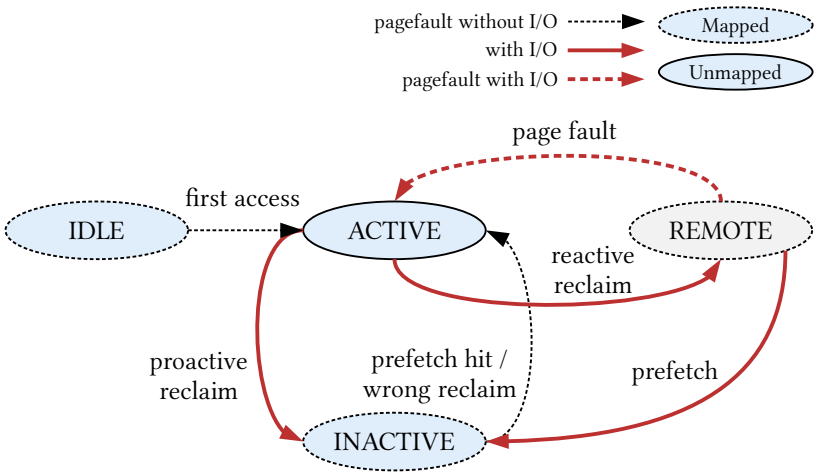


Figure 4.6 The page state transition diagram.

page must be reclaimed, which requires deleting the mapping in the page table and writing the page to the remote memory. Finally, the reclaimed page is restored with the original data by reading the page from the remote memory. In total, *two* RDMA I/O is required to serve the slow path request.

Fastpath in pagefault handler. Figure 4.5 shows the fastpath in RackVM pagefault handler. The pagefault handler first locks the page and checks if there is an available free page in the `INACTIVE` pool. In fastpath, the pagefault handler successfully get a page from the pool and use it immediately without writing the data to remote memory. The pagefault handler restores the previous data of the page, then returns to the userspace.

4.2.2 State transtion of RackVM page

Figure 4.6 shows the life cycle and state transition of a RackVM page. The RackVM page is the internal unit of memory management, it determines the lock granularity and the minimum size of I/O. A RackVM page size can be a multiple of the system page size.

We classify a RackVM page into the following four categories: `IDLE`, `ACTIVE`, `INACTIVE`, and `REMOTE`.

All pages are in the **IDLE** state on creation. An IDLE page has no mapping to a local page; a pagefault is generated on the first access to an IDLE page. In response to the pagefault, local memory is allocated to the page, and RackVM modifies the page state from IDLE to ACTIVE.

An **ACTIVE** page has a mapping in the page table; access on an ACTIVE does not generate pagefault. When a new ACTIVE page is created, it is inserted to the tail of ACTIVE list, and ACTIVE pages in the list are reclaimed from the head under memory pressure. We do not reorder the pages in that list to maintain least-recently-used (LRU) order. Rather we use a FIFO queue that avoids the overhead of sorting the list. While maintaining exact LRU order is beneficial for slow backends where the sort overhead is hidden by the I/O latency to access the device, for fast backends such as InfiniBand RDMA, maintaining LRU order would cause a bottleneck.

From **ACTIVE** to **REMOTE** transition occurs on the reactive reclamation (the slow-path in Figure 4.4). The allocated memory for the page is reclaimed and reused to serve a new ACTIVE page. The number of this transition should be minimized since the cost of the state transition is high and occurs in the critical path of the pagefault handler.

From **ACTIVE** to **INACTIVE** transition occurs on the proactive reclamation (Section 4.3) This transition occurs in the background to keep the INACTIVE pool populated. INACTIVE pages can be reused without writing the data to remote memory. By maintaining enough pages in the INACTIVE pool, we can reduce the overall pagefault handling latency. Rarely, the reclaimed page in INACTIVE list is accessed again. We call this case "wrong reclaim," which should be minimized. But the penalty of the wrong reclaim is not significant because the inverse transition can be made without an I/O.

From **REMOTE** to **ACTIVE** transition occurs when the VM accesses a remote page which is the case, the page has no mapping in the page table. Handling this pagefault has a high cost due to the I/O overhead to restore the original data; a prefetch technique can be deployed to reduce the number of this transition.

From **REMOTE** to **INACTIVE** transition is occured by the prefetch mechanism in RackVM (Section 4.3). Pages that are likely to be accessed in the near future are prefetched, then the pages are inserted to the head of INACTIVE list. If a prefetched

page in the `INACTIVE` list is accessed before it is reclaimed, from **`INACTIVE`** to **`ACTIVE`** transition occurs. In this case, we can serve the access request without performing an I/O. Increasing the number of prefetch hits is also an important optimization goal in RackVM.

4.3 Latency Hiding Techniques

The two main techniques for the major performance improvement in RackMem are *prefetch* and *proactive page reclamation*. We describe the detailed design of the two techniques in this section.

Prefetching. Prefetch can eliminate a costly I/O operation in the critical path of page fault handling. By prefetching pages that are expected to be accessed soon, we give more computation time to applications by reducing the number of page faults. Prefetch is especially useful when an application has clear sequential memory access patterns since multiple I/O operations on the consecutive addresses can be merged and processed much on many storage devices (e.g., HDD). Prefetch has been successfully employed in many I/O-oriented applications. Local memory is not scarce in the common scenario of such applications. Fetching additional data from I/O device in idle pages is cheap but gives significant benefit when the application has strong spatial locality. However, in remote memory paging scenarios, paging occurs when local memory is scarce, making unique challenges. Since local memory is scarce, we need to reclaim a few pages to serve prefetch pages. Prefetch should only evict cold pages that will not be accessed in the near future and correctly fetch data that is likely to be accessed soon. The cost of wrong prefetch is significantly higher in scare memory scenarios.

RackMem deploys a conservative approach for prefetch. RackMem strictly identifies sequential memory access patterns and performs prefetch only when an application shows strong sequential access patterns. The cost of wrong prefetch quickly offsets the benefit of prefetch, so we design our prefetch algorithm to do nothing most of the time to avoid being wrong. To better identify a more sequential access pattern for the prefetch, RackMem maintains per region and per-core prefetch window. It helps to separate mixed sequential memory access patterns from the stream.

Reactive and proactive page reclamation. When the local cache is full or a memory region hits the local memory limit, accessing an unmapped page requires paging out of a victim page. We summarize the main challenges for the page reclamation: first, when the victim page is dirty, paging out requires adding significant overhead to the memory access of the user application. Second, selecting *wrong* page as victim page (i.e., a page that is accessed soon again) will severely hurt the performance since it triggers a number of I/O to move data between the local memory and the backing storage. Third, since our target environments use high-performance backing storage (e.g., RDMA), software overheads of victim selection algorithms easily result in a significant performance loss. A good page reclamation algorithm needs a careful tradeoff between the accuracy and the computation overhead of the algorithm.

RackMem employs both reactive (foreground) and proactive (background) page reclaim policies to address the challenges mentioned above. Reactive page reclamation occurs when the local memory is full, and a page needs to be reclaimed to serve a page fault. Reactive page reclamation occurs in the critical path of the page fault handling, and we cannot hide the overhead in the background. The reclamation algorithm should quickly find a victim page rather than sacrificing the accuracy. Proactive page reclamation prepares a pool of immediately usable pages in the background, so it contributes to reducing the chance of reactive page reclamations and improves the overall remote paging latency. Since proactive page reclamation runs in the background, we can hide the latency from the critical path of the page fault handling. A complex but more accurate victim selection algorithm is preferred in proactive page reclamation.

4.4 Implementation

We implement RackVM as a separate kernel module of Linux kernel. Users can allocate and release RackMem managed memory address space through `mmap` system call on RackVM character device at `/dev/rack_vm`. We also provide a wrapper library written in C to facilitate writing userspace applications. Users can set region-specific memory management policy through the `debugfs` interface with UNIX file I/O operations. RackMem manages the regions that belong to the same process as a group and

manage them in the same subdirectory of the `debugfs`. We implement a userspace daemon that implements a simple local memory balancing policy to the co-located applications to show the effectiveness of the `debugfs` interface for the limited local memory management.

4.4.1 RackVM kernel module

RackVM kernel module implements a character device and registers it at `/dev/rack_vm`. Applications request RackMem virtual region by invoking `mmap` system call with a file descriptor of the character device and the desired size of the allocation. RackVM implements its own page fault handler for RackVM regions to capture memory accesses on the region and handle the remote memory paging transparently to user applications.

Prefetch algorithm. The prefetch algorithm in RackVM implements a modified version of the virtual memory area-based (VMA-based) prefetching technique that has recently been integrated into the Linux kernel. VMA-based prefetching is exploiting 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, hence VMA-based prefetching is considered a better choice.

RackMem maintains a dynamic window size that dictates how many pages are prefetched after a page fault. Other than the VMA-based prefetcher, RackMem tracks page faults 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 page fault (Figure 4.3). 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 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 page fault. 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. A page's data structure contains an 8-bit value that represents a metric for the 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). By interpreting the value 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. 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

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

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 $t = 0$.

4.4.2 Dynamic rebalancing of local memory to multiple VMs

RackVM provides a user-level API that allows to dynamically adjustment the local memory cache size of RackVM memory regions. In addition, statistics about RackVM such as the access frequency of each page or the total number of pagefaults are exposed to user-space through RackVM'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 RackVM memory regions in proportion to the number of active pages and pagefaults of a region. This memory re-balancer is especially useful when multiple VM requires fluctuating memory demands over time.

4.4.3 RackVM for virtual machines

The `/dev/rackmem` device node provides a simple interface to utilize RackVM-backed distributed memory, however, applications need to be modified to allocate memory from RackVM. For native applications, library interpositioning is used to intercept calls to the dynamic memory management library and are elaborated in Section 4.4.4 below. To seamlessly support Infrastructure-as-a-Service (IaaS) virtualization where workloads are executed in isolated virtual machines [13], RackVM includes a modified QEMU/KVM hypervisor. The RackVM QEMU/KVM hypervisor maps the memory of a virtual machine to a RackVM memory region instead of local memory. The modification requires only a few additional lines of code that open RackVM's `/dev/rackmem` device and pass its descriptor to the (already present) `mmap()` system call to setup the memory of a VM.

4.4.4 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, RackVM 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 RackVM region that serves as the heap area for the numerous and often small dynamic memory requests. This avoids interacting with the lower levels of RackVM and is a design also chosen by related work [12].

Chapter 5

RackMem Distributed Virtual Storage

RackMem Distributed Virtual Storage (RackDVS) provides a byte-addressable storage abstraction of multiple storages in remote nodes, and it is also the main storage backend of RackVM (Chapter 4). Figure 5.1 shows an overview of RackDVS. RackVM request virtual byte-addressable storage from RackDVS. The virtual storage address space is linearly mapped to the virtual memory address space; an evicted page is stored at the same offset in the virtual storage in the result of paging.

We use *remote memory* as the primary storage backend for RackDVS to realize a rack-scale memory disaggregation. By attching RackDVS to RackVM, the evicted pages in RackVM are automatically distributed to the cluster by RackDVS.

RackDVS also supports fault tolerance of remote memory backend; An evicted page is optionally replicated to the local storage for reliability. On the failure of a remote node, RackDVS restores the same copy from the local storage.

Another interesting aspect of RackDVS is multiple storage backends support which means RackDVS is not limited to the remote memory backend. Any storage device that implements the RackDVS interface can be registered as a backend of RackDVS. RackDVS utilizes multiple storages with a simple priority-based policy.

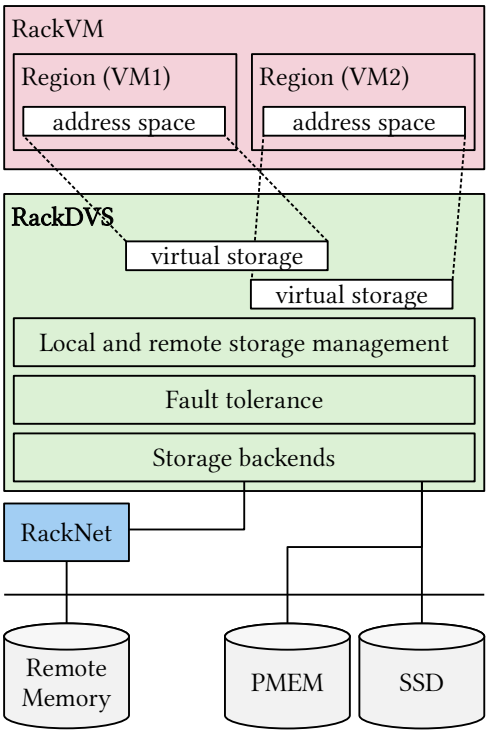


Figure 5.1 Overview and the main components in RackDVS.

In the rest of this chapter, we explain each feature of RackDVS in detail.

5.1 The Distributed Storage Abstraction

RackDVS provides an abstraction of byte-addressable linear address space on back-end storage devices. RackDVS abstraction is similar to the block device in Linux but provides close to hardware performance of modern fast storages (e.g., RDMA backed remote memory). Once space is allocated with RackDVS, I/O requests on the space are directed to hardware without a complex software-based I/O scheduling.

RackDVS manages the address space at slab granularity that is usually much larger than a RackVM page size (Figure 5.2). The coarse-grained space management reduces the metadata overhead in large-scale space allocation.

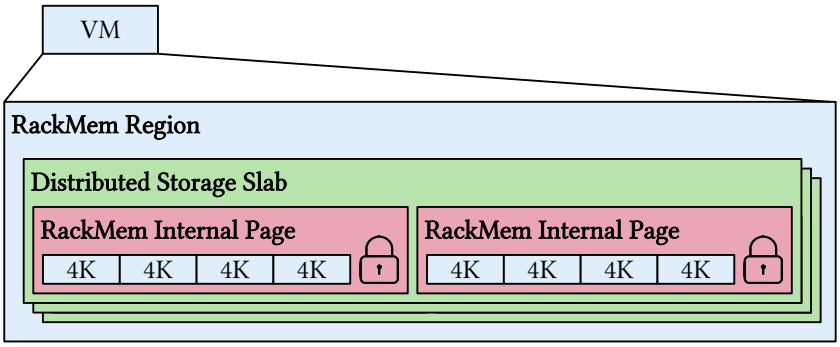


Figure 5.2 Data layout of RackMem.

5.2 Memory Management

Like RackVM, RackDVS regions are identified by a descriptor 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. RackDVS manages distributed storage in fixed-size slabs which can store a number of RackMem internal pages (Figure 5.2). A slab is allocated when its data is accessed for the first time; the *lazy* allocation of slab reduces the total memory footprint of RackDVS by minimizing the number of slab allocations.

5.2.1 Remote memory allocation

RackDVS selects a donor node from the list of available nodes providing backend storage using the "power of two choices" distributed load-balancing algorithm [50] that has shown good results in practice [28]. The algorithm randomly picks two nodes from the node list and selects one with more idle memory.

5.2.2 Remote memory reclamation

By reading a page from the remote memory, two same copies of a page are generated on both the remote memory and the local memory. The remote side copy only wastes the memory, and ideally, it should be released for other use. Thanks to the single owner assumption in RackMem (there is only a single user of a RackVM region at a

time), the remote memory can be safely released when the same copy is available in the local memory.

To reclaim the remote memory, RackDVS deploys a reference counter-based remote memory reclamation mechanism. On writing an internal page to the slab, RackDVS increases the reference counter of the slab. If the slab accommodates n internal pages at maximum, the reference counter can be increased up to n. Inversely, reading an internal page from the slab decreases the reference counter. When RackVM reads all pages in a slab and the reference counter goes to zero, RackDVS reclaims the slab so that the remote node can use the released memory.

5.3 Fault Tolerance

RackDVS mainly utilizes the remote memory backend, which provides a single-digit microseconds latency for fetching a page. The high performance and low latency give significant benefits for utilizing remote memory. However, the benefit does not come with free. By using remote idle memory in multiple nodes in the cluster at the same time, also increases the chance of failure. If a node goes offline due to failure, the nodes using the failed node cannot access the data anymore. The result will be catastrophic by cascading the failure throughout the cluster. In this scenario, the failure domain is extended to the cluster level by using RackMem.

RackMem provides fault tolerance by duplicating the write I/O to the local storage. By duplicating write requests to the local storage, RackDVS can access the data in the case of remote node failure.

5.3.1 Fault-tolerance and Write-duplication

The write-duplication enables fault tolerance by making the data survive on the failure of a remote node. There are three approaches for duplicating the write request to the local storage. We discuss pros and cons of each approach as follows.

Consecutive synchronous I/O to remote and local storage. In this approach, RackDVS first writes to remote memory and writes to the local storage after the completion of remote memory write. The main advantage of this approach is the

simplest design and implementation. It does not require launching a separate thread for asynchronous I/O and does not need to handle I/O completion from outside the I/O function. However, the I/O latency is significantly increased by delaying the completion of I/O to wait for the completion of local storage write. We also lose the all performance benefit of using remote memory with this approach.

Write synchronously to the remote memory and asynchronously to the local storage. In this approach, RackDVS starts non-blocking write to the local storage and immediately starts remote memory write. Once the remote memory write is completed, the `write` function returns without waiting for the completion of local storage write. The local storage write is handled in a separate thread in the background.

This approach provides minimal write latency among the three approaches; the latency is almost the same as the original remote memory write. Infiniswap [28], a state-of-the-art remote paging system, also deploys this approach for their fault tolerance mechanism.

However, this approach has a few critical shortcomings. First, the `write` request in RackDVS mostly occurs in the context of local memory reclamation. It is expected that the memory that having the data for the `write` request will be immediately released after the completion of the `write` request. However, the memory cannot be released until the asynchronous local write is ongoing. Second, this approach requires a dedicated background thread for the local storage write and handling the completion. It adds non-negligible overhead to the system, which can degrade the overall performance.

Write asynchronously to the remote memory and synchronously to the local storage. is bounded to the local storage performance. Figure 5.3 illustrate the flow of write duplication in this approach. RackDVS starts non-blocking remote memory write, then starts local storage write immediately and waits for the completion of the local storage writes. The non-blocking remote memory write is also handled inside of the same `write` call after the completion of the local storage write. In most cases, remote memory write completes earlier than the local storage write; consequently, the `write` latency is bounded to the local storage performance.

This approach can exploit the full benefit of the I/O model of RDMA and the

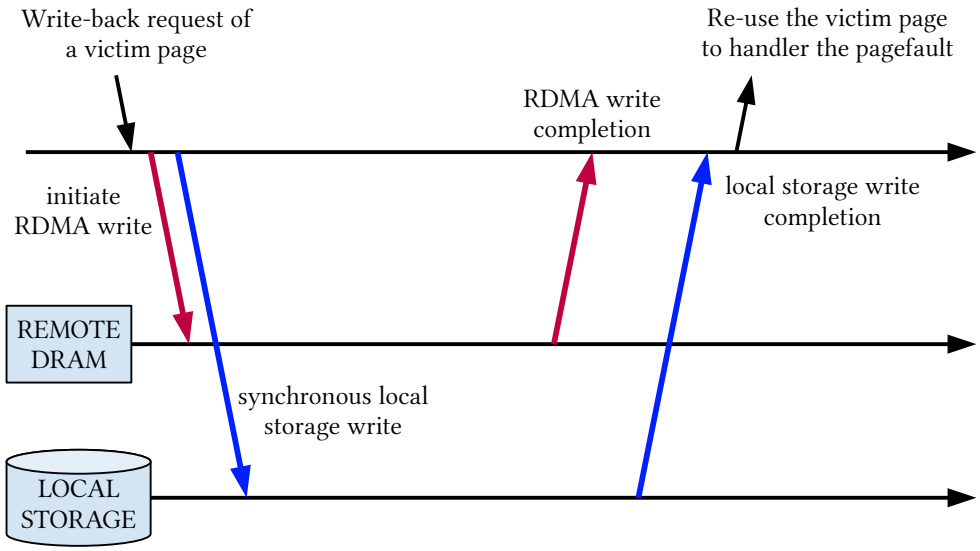


Figure 5.3 Write duplication for fault tolerance in RackMem.

proactive page reclamation of RackVM. First, posting an I/O request in the RDMA model is naturally non-blocking; a background task for the non-blocking I/O is unnecessary in RDMA. Second, `write` occurs in the context of the page reclamation in RackVM. Since RackVM pro-actively reclaims pages in the background, we can hide the local storage write latency in the background. This approach has the most simple design among the three approaches with little degradation on the overall performance.

We deploy this approach for the main fault tolerance mechanism in RackDVS. We discuss the overhead and requirement of storage performance for the transparent I/O duplication in Section 8.3.3 of Chapter 8.

5.4 Multiple Storage Support in RackMem

RackMem supports multiple storages and builds a hierarchical storage by utilizing them with priority. Figure 5.4 shows the overall design of the hierarchical storage using multiple backend storages. Different storage types of slabs can support a RackDVS

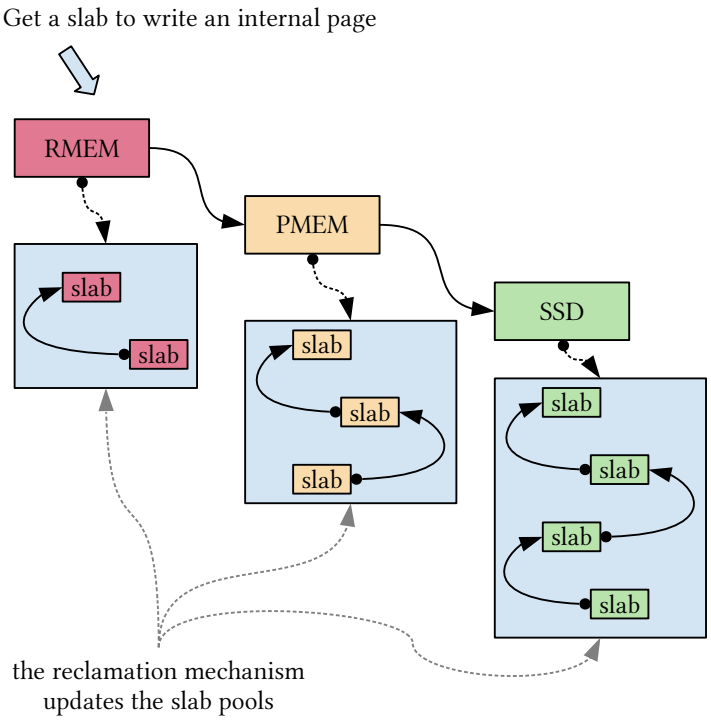


Figure 5.4 Multiple storage backends in RackDVS.

region. When a region user accesses an unallocated space in the region, RackDVS allocates a new slab from the hierarchical storage by checking the remaining space of each storage tier. The user assigns a priority for each storage; a slab is allocated from the highest priority storage that has a remaining space to satisfy the request.

In RackMem, the slab allocation requests are mainly performed to serve and write I/O of page reclamation. By allocating a slab from a faster device, we can reduce the overhead of page reclamation.

The slab of faster storage is frequently updated with the page reclamation mechanism, which maximizes the utility of the hierarchical storage.

5.5 Implementation

We implement RackDVS and backend storages as separate kernel modules. The RackDVS kernel module provides byte-addressable storage with a set of storage APIs that support allocation, deallocation, and data transfer operations.

RackDVS defines I/O interface for backend storage. Therefore, any device driver that implements the RackDVS interface can be used backend storage of RackDVS. This section also discusses the main backend storages using remote memory and other storage backends.

5.5.1 The Remote Memory Backend

We implement the remote memory backend using RackNet (Chapter 6), a kernel space RDMA and the RPC library of RackMem. RackDVS registers RPC functions for allocation, release, and memory management on top of RackNet.

The allocation handler receives a remote slab allocation request with a size. The handler is executed in kernel space and allocates a physically contiguous memory for RDMA, and then sends the DMA address to the client side so the other machine can store evicted page with RDMA.

Reducing the slab allocation overhead. The remote memory backend needs to issue an RPC request to allocate a slab. The slab allocation overhead is significantly higher in the remote memory backend than local storage backends due to the roundtrip latency; optimizing the slab allocation latency is the main implementation goal in the remote memory backend. To minimize slab allocation latency, we make the remote memory backend implement a local slab pool to serve most incoming requests without the roundtrip to the remote node.

The slab pool has a number of slabs that are immediately usable by RackDVS. The slab pool has three configurable parameters. The *target pool size* defines the desired size of the pool. The *low* and *high* thresholds define the threshold to trigger a background task to resizing the pool. The backend device checks the pool size with the given thresholds on every slab allocation request and launches a background task if resizing is required.

With this optimization, the backend device can respond to allocation requests immediately in most cases.

5.5.2 Linux Demand Paging on RackDVS

RackDVS layer can also serve as a storage backend to Linux's virtual memory manager. It is useful when virtualization support is not available in the system. By utilizing Linux's virtual memory, we can bring the benefit of memory disaggregation to all processes not limited to virtual machines with sacrificing the paging performance.

We modify the implementation of Linux's `nullb` high-performance block device [26] to employ RackDVS 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 [28] 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.

Chapter 6

Networking

The RackMem networking layer (RackNet) provides an easy abstraction of inter-node communication mechanisms such as remote procedure call (RPC) and remote direct memory access (RDMA) for other components in RackMem. RackNet provides good scalability by efficiently utilizing the limited on-device resources with connection sharing, reduces RPC overhead handling with an interrupted-based completion handling mechanism.

6.1 Design of RackNet

Figure 6.1 illustrates the overall architecture of RackNet. RackNet establishes only two QP connections between two nodes. We dedicatedly use each QP for RDMA and RPC request handling separately.

Each QP serves all RDMA and RPC requests from other RackMem components such as RackDVS and RackVM. The remote memory backend in RackDVS utilizes the RDMA QP for reading/writing data from/to remote nodes and utilizes the RPC QP for allocation/deallocation of remote memory. RackVM also uses the RPC QP to implement the instant region migration in Chapter 7.

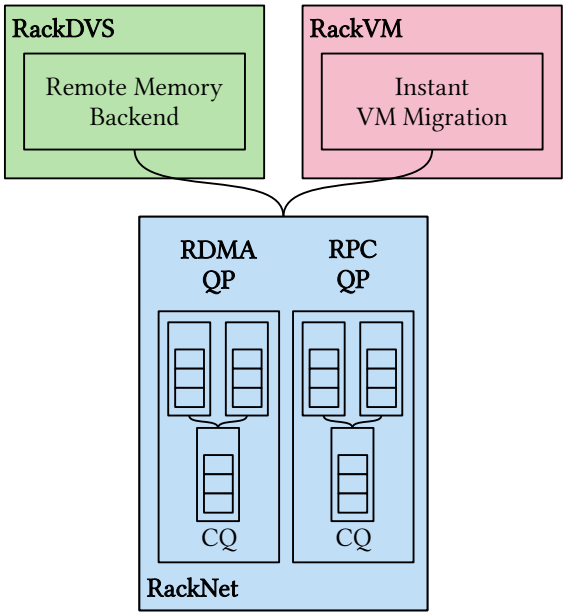


Figure 6.1 The RPC message layout of RackNet.

6.2 RackNet RPC Implementation

We implement RackNet as a separate kernel module. RackNet exposes userspace interfaces for connection and disconnection of nodes. Useful statistics of RackNet is also available through `debugfs`. Once a connection is established with other nodes, kernel-space applications can use RackNet APIs to register and call remote functions.

6.2.1 RPC message layout

Figure 6.2 shows the RPC message layout of RackNet. A message has seven reserved entires, which are in total 52 bytes. An arbitrary size payload is included at the tail; the size is RPC implementation-specific.

`RPC_ID` entry defines the function ID of a RPC handler. `RPC_TYPE` presents the side where the message is processed, either client or server. `SEND_COMP` is the flag to check the send request completion. `RECV_COMP` is the flag to check the receive request completion. `RET_CODE` provides information of success of failure of the request.

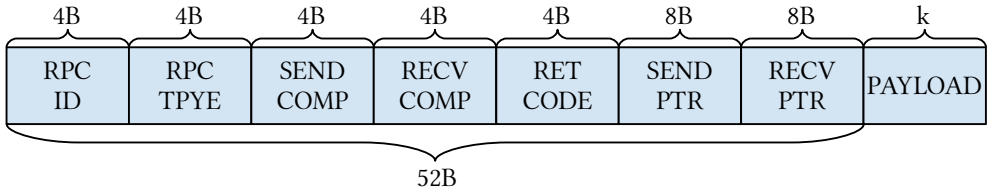


Figure 6.2 The RPC message layout of RackNet.

`SEND_PTR` is pointing the address of the send message. `RECV_PTR` is pointing the address of the receive message. `PAYLOAD` contains the input arguments or the result of an RPC function.

We explain how each field is used in the context of RPC processing in the next subsection.

6.2.2 RackNet RPC processing steps

Figure 6.3 shows an RPC processing timeline of RackNet. First, the client allocates a new RPC message and fills the fields and the payload with corresponding data (❶). In the next step, the completion handler on the client-side is waked up and process the completion by setting `SEND_COMP` field in the message. As a result of the completion handling, the client receives notification by polling the `SEND_COMP` field in the message (❷ and ❸).

Now the server-side has the received message from the client. The completion handler on the server-side launches a new thread and passes the message to process the RPC request (❹). As the next step, the RPC handling thread processes the request and sends the result to the client by making a new message. The RPC handling thread checks the `SEND_COMP` field by polling to check the success of message transmission. (❺, ❻, ❼, and ❽)

The client-side handler receives the RPC result message from the server-side. The client-side handler reads `SEND_PTR` field in the received message to find the original request message and then sets `RECV_PTR` field in the original message with the corresponding pointer. Finally, the client-side handler notifies the client by setting

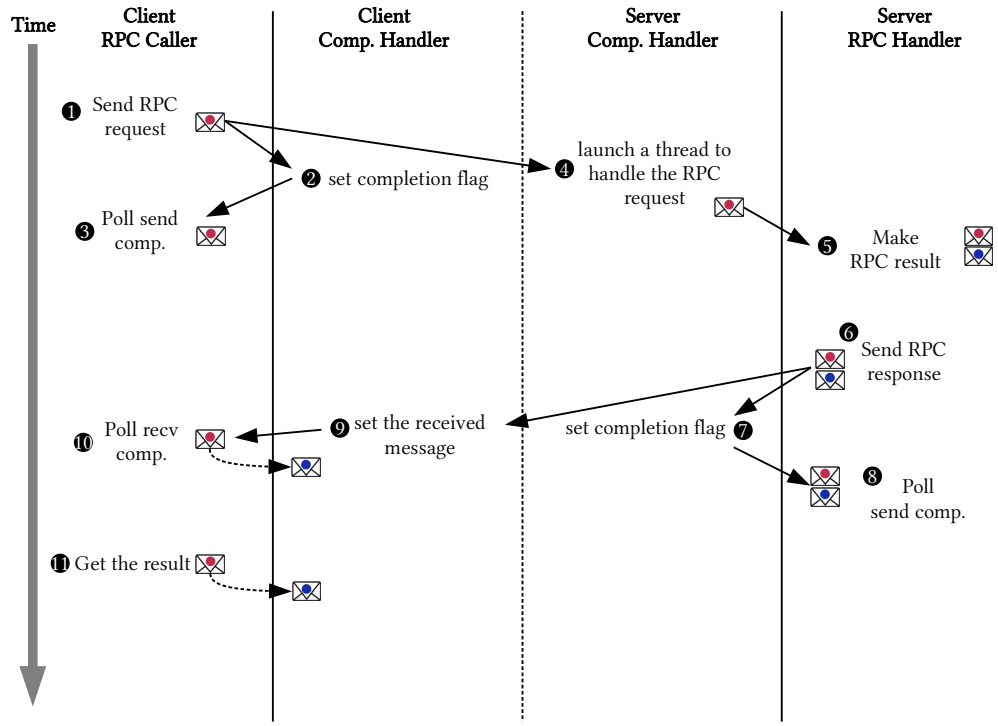


Figure 6.3 RackNet RPC processing timeline.

`RECV_COMP` field in the original message (❾ and ❿).

This RackNet RPC implementation achieves 17μs of latency on average for processing a dummy RPC function. The latency number is slightly higher than state-of-the-art RPC systems [37, 54, 65], but it does not limit the performance of the current RackMem implementation. We have optimized RackMem to hide RPC processing latency in the background and reduce the number of RPC calls in common scenarios by using a special function to process bulk requests in one RPC. The current design and implementation of RackNet provide more than enough performance and great usability.

Chapter 7

Instant VM Live Migration

In this chapter, we present INSTANT, a novel VM live migration technique that *instantly* and *seamlessly* migrates VMs running on RackMem. The proposed technique (almost) instantly migrates a VM by only transferring a small amount of meta-data containing the VM's memory pages to the destination instead of sending the entire data to the destination node.

INSTANT employs a number of techniques to minimize the performance degradation incurred immediately after migration by proactively bringing recently accessed memory pages into the local memory of the destination host. A thorough evaluation with batch-oriented and latency critical applications shows that INSTANT outperforms existing techniques by a wide margin and achieves close to constant migration times and quick performance recovery after migration.

Instant VM live migration address a slightly different problem that we have discussed so far. We start this chapter by introducing the necessary background to understand the rest of this chapter better.

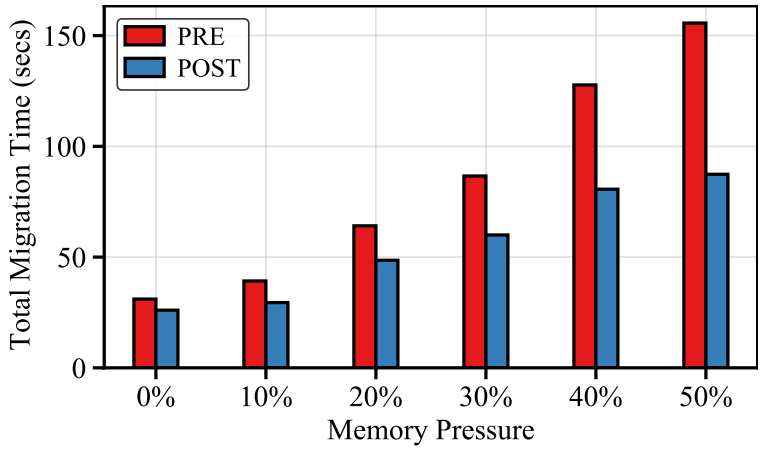


Figure 7.1 Total migration time of VM running YCSB workloads under memory pressure.

7.1 Background and Motivation

7.1.1 The need for a tailored live migration technique

As we have outlined in the introduction, the combination of disaggregated memory with virtualized environments is an ideal match to solve intermittent resource spikes by moving infrequently used pages to a less-loaded remote machine [36]. If the resource shortage continues, however, one or more VMs from the overloaded host need to be migrated away. Existing live migration techniques are ill-suited for this purpose since they copy the entire memory from the source to the destination node, i.e., in a high-resource-utilization phase with memory already paged out, these algorithms *increase* memory pressure during migration because they need to bring in all remotely storage pages back to the source host before sending the data to the destination.

Figure 7.1 shows the total migration time of a VM running the YCSB [20] workloads under memory pressure. We insert 5 million keys to a Redis database running in the VM and send requests with the *update mostly* workload in YCSB benchmark. The requestor client runs in a separate machine and uses 20 threads at full speed to generate the requests. We measure the total migration time of the pre-copy (`PRE`) and post-copy (`POST`) live migration technique in QEMU [56]. To simulate memory pressure, we limit the VM's local memory using `cgroups` [25] with a remote memory swap backend [28].

The results show a significant degradation in the total migration time under memory pressure. In the unloaded case (all memory local), PRE and POST finish migration in 31 and 26 seconds, respectively, while transferring 15.6 GiB at 514 MiB/s and 11.3 GiB at 447 MB/s at separately. As the memory pressure increases, the throughput drops rapidly. At a memory pressure of 50% (50% of memory is remote), the total migration time of PRE and POST increases by a factor of 5.0x and 3.4x. The reason for this slowdown is that the pages need to be brought back to the source machine before being sent to the destination; and these transfers compete with the migration for network bandwidth.

We need a live migration algorithm that is aware of the underlying memory organization and that does not needlessly copy remote pages via the source to the destination, but instead only transfers the *memory location* such that the destination node can retrieve the page when it is accessed.

7.1.2 Software bottlenecks

Live migration support in popular open-source hypervisors [5,56] was initially developed over a decade ago when 1 Gbit/s network was standard. As a consequence, the implementations cannot utilize the full bandwidth of modern networks, and throughput stagnates at the 10 Gbit/s level. In addition, optimizations that were effective with slow networks have no or an adverse effect as the network is getting faster. I/O is much cheaper these days, and using computation to reduce I/O is in general not recommended anymore. For these reasons, we believe it is the right time to architect VM live migration and make develop a novel instant live migration algorithm that fully exploits the benefits of remote memory.

7.1.3 Utilizing workload variability

VM live migration is a widely known technique for improving resource utilization by moving VMs from highly loaded to lightly loaded servers. Modern workloads exhibit a high variability in their resource usage; resources are frequently idle over the entire execution period. VM live migration should be completed fast enough to utilize these resource holes. For example, idle resources on the destination server may have disappeared by the time a long-running VM live migration has completed. The presented

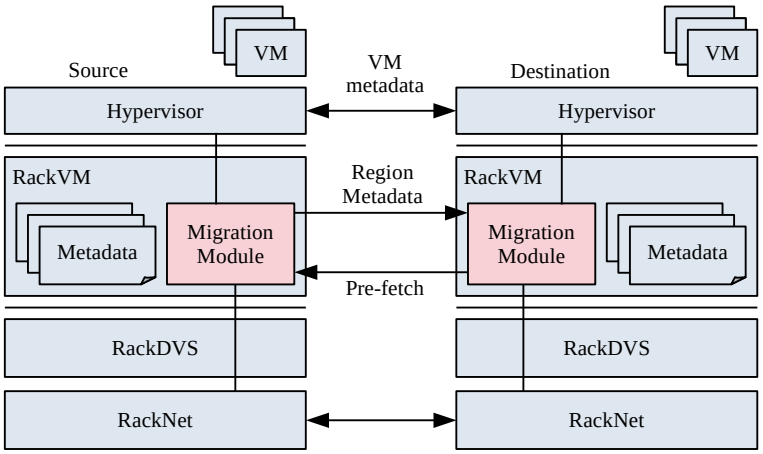


Figure 7.2 Major components in INSTANT.

instant VM live migration techniques enables new opportunities for better utilizing resource variability in data centers. The ability to complete VM live migration around 100ms will make many chances for utilizing the idle resources in data centers.

7.2 Design of INSTANT

Figure 7.2 shows the high-level architecture of INSTANT. The migration module handles *region* migration request from users. On a migration request, the module communicates with its counterpart on the destination to initiate a copy of the region metadata and invalidate the region on the source side. Region migration transfers only a minimal amount of data, this is what enables the *instant* part of the presented VM migration technique. The hypervisor heavily relies on this module to perform instant VM migration.

7.2.1 Instant Region Migration

INSTANT migrates a VM by only sending the metadata of the VM's virtual memory. In other words, INSTANT only has a *stop-and-copy* phase during migration. It is the main difference to other approaches that require at least two phases, "prepare + stop-and-copy" for *pre-copy* and "stop-and-copy + restore" in the case of *post-copy*. In addition

to that, *pre-copy* and *post-copy* require a copy of the entire memory to complete a VM migration.

Despite of the *instant* migration capability, a VM's performance can be degraded after migration since the working set pages do not exist in the destination's local memory. We implement an optional *restore* phase to quickly restore VM performance after the instant migration through a background fetch process.

Stop-and-copy. The instant VM migration immediately starts from this pahse. VM stops execution, and the necessary data to resume the VM on the destination is transferred. The virtual memory module sends the region metadata to the destination, and the hypervisor sends the minimal VM state and the VM's metadata to the destination. The region metadata includes the location and the address of each page. The VM state includes the processor state, device buffers, and the VM configuration.

Restore. The VM continues its execution on the destination node. The pages belonging to the VM's working set will still be located in the source machine. Depending on the number of working set pages and the VM's activity, the VM can experience a severe performance degradation on the destination. To quickly restore the performance of the VM, we optionally launch a background task that pre-fetches the working set pages from the source machine. The background task starts with the list of pages in the active list, and fetches the pages in recently added page first order.

7.3 Implementation

We implement Instant components as a set of Linux kernel modules. Instant comprises two main modules. First, the virtual memory module provides transparent paging to disaggregated memory with the region migration mechanisms. Second, the networking module provides RPC and RDMA functionalities to the virtual memory.

We deploy QEMU/KVM for virtualization and modify it to utilize Instant for VM's memory allocation.

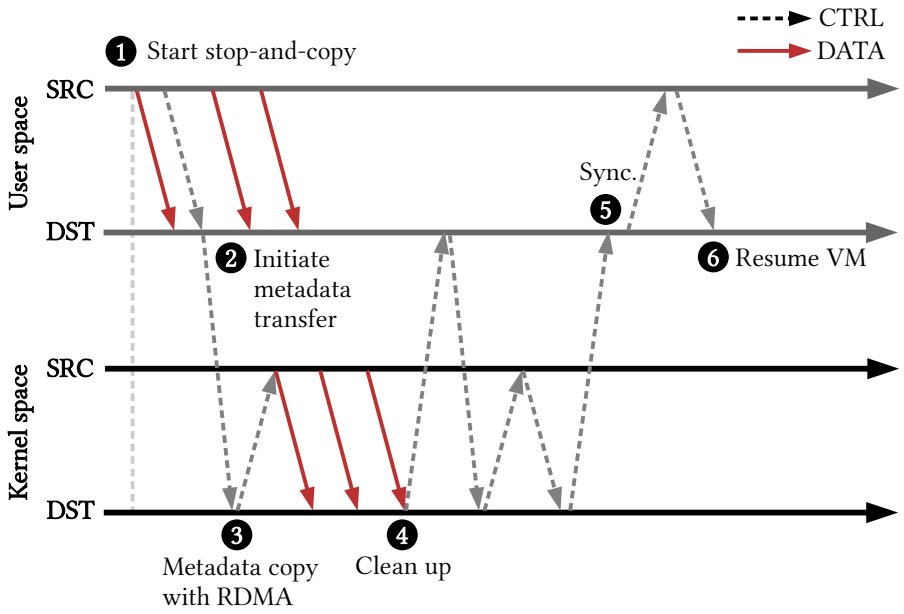


Figure 7.3 Instant VM live migration timeline.

7.3.1 Extension of RackVM for INSTANT

RackVM module allocates a unique region identifier (`rID`) for each region, a remote node can access the region with `rID` through the associated RPC call provided by RackNet.

The kernel module implements a set of `ioctl` calls to provide the region migration mechanisms. There are two `ioctl` calls for *stop-and-copy*, and *restore*. The optimizations are discussed in more detail in Sections 7.3.4 and 7.3.3.

7.3.2 Instant region migration

Figure 7.3 shows breakdown of each step of *stop-and-copy* in INSTANT.

1. QEMU stops the VM on the source machine and starts the stop-and-copy phase. The source side QEMU sends a small message including `hostname` and `rID` to the destination to starts region migration. The region migration does not block the stop-and-copy in QEMU. The two data streams send data in parallel

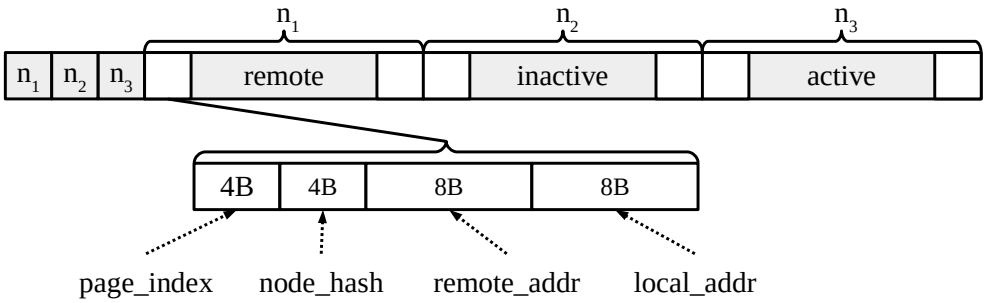


Figure 7.4 The metadata layout.

throughout the stop-and-copy phase.

2. The destination receives region migration requests and issues the associated `ioctl` call to the kernel module with the given parameters from the source. The kernel module find the source node using the `hostname` and request metadata migration with the `rID`. This communication occurs in the kernel space in parallel to the userspace communication.

3. The source-side kernel module allocates memory space for RDMA and fills out the space with the metadata of the target region. Figure 7.4 shows the metadata layout. The space is mainly divided into three parts for remote, inactive, active pages separately. The size of a page metadata is 24B. The first 4-byte is the page index. The next 4-byte is the host node of the page. The next 8-byte is DMA address for RDMA, remote node can access this page using this address. The last 8-byte is the local virtual address, local node can directly access the page's data using this address. The destination receives the DMA address of the metadata and reads it with one RDMA read.

4. The destination kernel module reconstructs the region using the received metadata and releases the remote memory of metadata by issuing an RPC request to the source machine. Now the control is returned to the userspace to complete the migration.

5. The destination side QEMU synchronized with the source and resume the VM.

7.3.3 Pre-fetch optimizations

Running a background task on the destination host to fetch working set pages from the source machine is an effective technique to reduce the burst page faults in the restoration phase. When the hypervisor enables this optimization, the destination task launches a kernel thread to read the recently used pages from the source machine. The destination node can identify the list of pages for prefetch by investigating the received region metadata. The received metadata reserves the least recently added order of the pages. The region migration module inserts the pages in the reverse order to the prefetch recently accessed pages first.

7.3.4 Downtime optimizations

Instant VM live migration sends the region metadata in *stop-and-copy* phase in addition to the CPU states, device buffers, and VM metadata which makes its downtime longer depends on the size of the metadata. In this section, we present a number of optimization techniques to hide the metadata transfer latency.

Latency hiding. Instant VM live migration only sends the memory region's metadata through the kernel-to-kernel connection. The rest of the data, such as CPU states, device buffers, and VM metadata, are transferred by the separate link by QEMU in the userspace. The CPU states, device buffers, and VM metadata are separately allocated, and each allocation is very small (usually less than 1MB in total). The data is naturally fit to the local memory use case and little reason to store them on the disaggregated memory.

However, preparing and sending small data chunks through the network is inefficient and takes a few milliseconds to complete. By considering the presented reasons, hiding the metadata transfer latency along with them is a reasonable idea.

We initiate the metadata transfer by calling an associated `ioctl` call to the region migration module. The function initiates a kernel task to transfer metadata to the destination and immediately returns to the userspace. After the completion of the metadata transfer, the source side issues an RPC call to restore the necessary data structure on the destination. On the while, the userspace process sends the rest and complete the VM migration.

7.3.5 QEMU modification for Instant

We modify QEMU/KVM to demonstrate the instant VM migration. The modified QEMU provides interfaces to configure migration capabilities (on/off switches) and parameters. Migration parameters allow fine-tuning of each capability with the set of configurable parameters. We add a new migration capability *instant* to QEMU with tuning parameters to control the migration mechanisms of the kernel modules.

Chapter 8

Evaluation - RackMem

This chapter presents an evaluation of RackMem on a real cluster with a wide range of latency-critical and batch processing workloads. We divide the evaluation into two main parts (Chapter 8 and Chapter 9). The first part focus on the evaluation of single or co-located application performance under memory pressure with RackMem. We mainly address the following questions throughout the first part evaluation.

- Can RackMem's virtual memory improve the pagefault handler throughput and reduce the tail latency? (Section 8.2)
- Does RackMem provide better performance for applications under a disaggregated environment? (Section 8.3)
- How efficiently can RackMem utilize local memory when co-located applications share local memory? (Section 8.3.7)
- Can RackMem improve performance of a job processing cluster by sharing unused memory between physical machines? (Section 8.3.8)

In the second part, we present evaluation of instant VM live migration on RackMem.

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 8.1 Evaluated RackMem configurations.

8.1 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 with KVM/QEMU 4.2 for virtual machines.

Effect of RackMem Optimizations. We evaluate RackMem with the four distinct configurations: RACK, RACK.R, RACK.P, and RACK.RP to evaluate the effect of individual optimization described in Section 4.3. Table 8.1 shows the detail configuration of each optimization.

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 8.2 gives an overview of the evaluated implementations. LMEM stores and retrieves data in the local DRAM. This approach uses pmem.io [55] 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 or Linux.RMEM 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 5.5.2) because the open-sourced implementation [32] does not run correctly on recent Linux kernels which also include patches that improve the performance of the swap subsystem [24].

Enforcing local memory limits and sharing. To evaluate performance with a varying local cache size, we artificially limit the amount of local memory available

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

Table 8.2 Compared implementations.

	Mean (30s)	Max (30s)	Mean (60s)	Max (60s)	Peak RSS
Spark.PgRank	947	3365	2104	4181	4372
Parsec.Bodytrack	553	1970	1198	2190	2234
Parsec.Canneal	382	1289	822	1730	1845
Parsec.Dedup	1064	4148	2416	4734	4832
Parsec.Raytrace	579	1696	1241	2389	2826
Parsec.Vips	932	3506	2105	4090	4178
NPB.bt	184	1336	415	1401	1401
NPB.dc	301	1295	663	1399	1400
NPB.ft	383	1871	881	1984	1983
NPB.is	393	1542	898	1736	1737
NPB.mg	288	1085	640	1147	1145
OLTPBench.tpcc	259	1318	488	1902	2715
OLTPBench.twitter	297	1346	567	2361	3421
OLTPBench.wikipedia	410	1514	766	2023	5199

Table 8.3 Working set size for 30/60 second windows and peak resident set size (RSS) of target applications. Values in MB.

to the applications. In the case of RackMem, the amount of local cache size can be dynamically set through the `debugfs` interface (Section 4.4). 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` [25] by setting the `memory_limits_in_bytes` parameter to the desired value.

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`) [67], CIFAR10

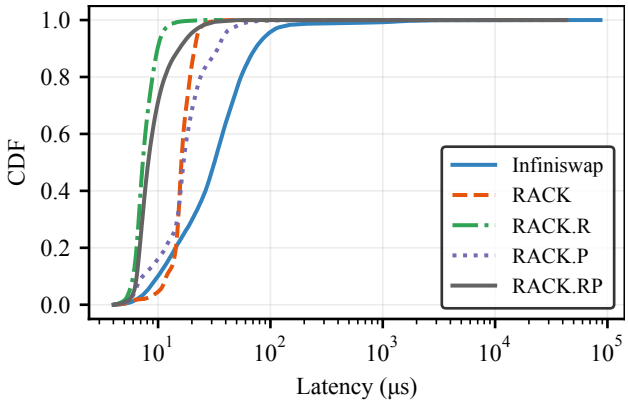


Figure 8.1 Pagefault handling overhead of RackMem.

inference from Tensorflow (`Tensorflow.CIFAR10`), four applications (`Canneal`, `Raytrace`, `Vips`, and `Dedup`) from the Parsec benchmark suite [18], and five workloads (`mg`, `is`, `ft`, `bt`, and `dc`) from the NASA Parallel Benchmarks (NPB) [17]. The applications `tpcc`, `twitter`, and `wikipedia` from OLTP-Bench [22] represent latency critical workloads.

Table 8.3 lists the benchmarks along with the mean and the maximum of the working set size for a sliding window of 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.

8.2 Pagefault Handler Latency

The latency of handling major pagefaults by RackMem and the Linux kernel is measured by adding tracepoints [21] 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 2.2. 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 $7\mu s$ at 50, 12 at 95, 17 at 99, and $35\mu s$ at the 99.9 percentile. Compared to Linux, RackMem

achieves an 80-times shorter latency at the 99.9 percentile.

Figure 8.1 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 8.1: 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 8.1).

8.3 Single Application Performance

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

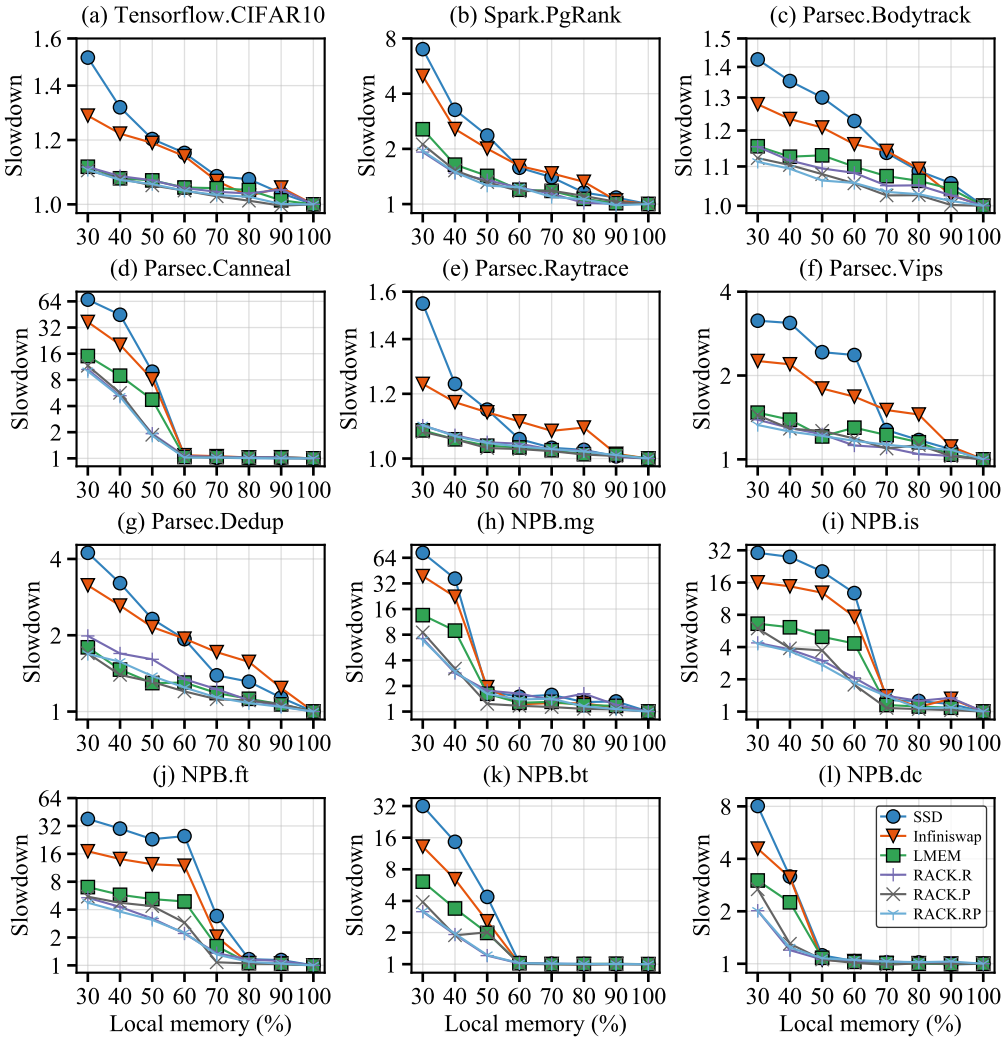


Figure 8.2 Normalized execution time of batch-oriented applications under memory limits (log scale, lower is better).

8.3.1 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.

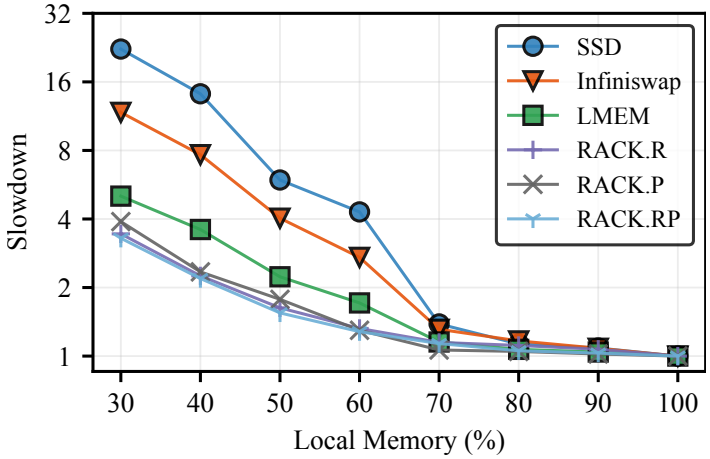


Figure 8.3 Averaged normalized execution time of all batch-oriented applications.

Figure 8.2 plots the normalized execution times of the batch-oriented applications under the different configurations. The results demonstrate that RackMem's optimized virtual memory directly 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.

The aggregated results over all batch-oriented applications in Figure 8.3 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.

8.3.2 Internal pagesize and performance

The internal page size in RackMem determines the granularity of locking and minimum I/O unit size. It is an important performance tuning parameter and analyzing the effect of the internal page size on performance is necessary for better optimization

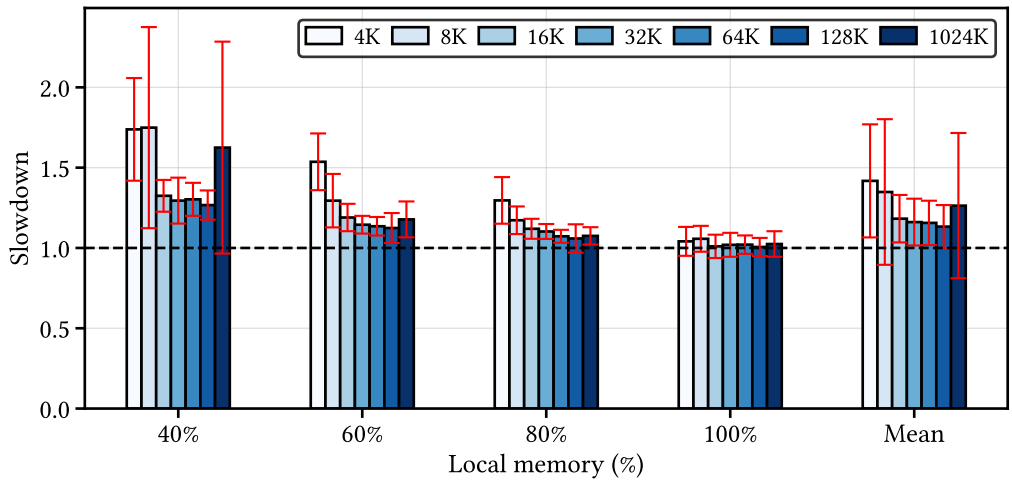


Figure 8.4 Effect of the internal page size.

of RackMem.

Figure 8.4 shows Spark.PageRank performance under various memory limits for internal page sizes from 4KB to 1MB. The result shows a performance improvement trend with increasing the internal page size from 4KB to 128KB. However, the amount of improvement is also diminishing with the increasing pagesize. A larger size than 128KB rather degrades the performance; the benefit of increasing the internal page size does not exist anymore.

Many modern data-intensive applications have spatial locality in memory access patterns. Such types of applications can benefit from a larger internal page size by fetching more data with one I/O read. However, the increasing internal page size, at the same time, increases the chance of lock contention on the same address space, which finally offsets the benefit of a larger page size.

8.3.3 Write-duplication overhead

RackMem supports fault tolerance by duplicating write I/O to the local storage. However, enabling write duplication also increases the I/O latency for write. Therefore, it is important to see the performance impact caused by the write-duplication.

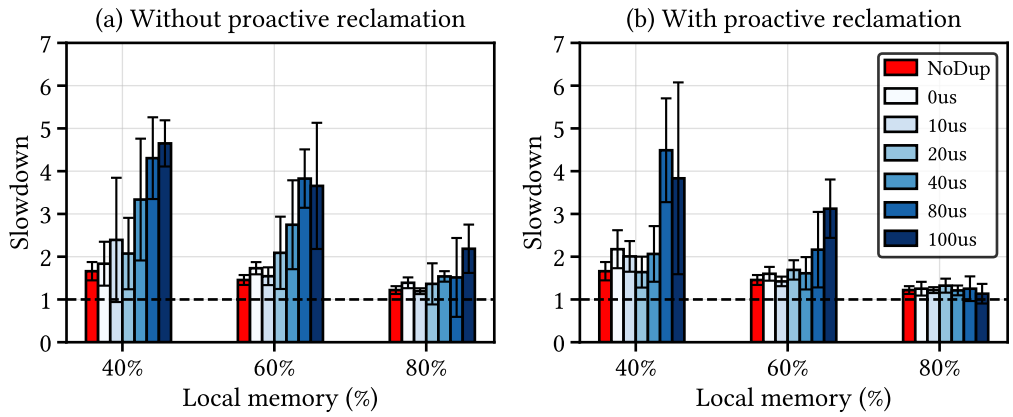


Figure 8.5 Write duplication overhead.

To measure the effect of I/O duplication and investigate the storage requirements for seamless I/O duplication, we implement an emulated local storage that using local DRAM as the storage. The emulated storage writes the data to the DRAM with an additional delay. We emulate the additional delay by adding busy waiting in the I/O function. Figure 8.5 shows the execution time of Spark.PageRank under local memory limits for 40%, 60%, and 80% separately. We also measure the effect of proactive reclamation by running the same experiments with and without the po-active reclamation.

Figure 8.5 (a) shows Spark.PgRank performance with I/O duplication, but with disabled proactive page reclamation. The performance degradation from the I/O duplication is clear in this result with the increasing delay in the emulated storage. The additional delay is directly contributed to the longer pagefault handling latency, which results in poor performance of the target application.

Figure 8.5 (b) shows the same experiments with enabled proactive reclamation. The application much better tolerates the increased I/O delay than the setup in Figure 8.5 (b). The proactive reclamation eliminates most I/O write in the critical path, and the optimization effect is clearly shown in the result. The result also implies a 20 microseconds latency for writing a page to local storage would not degrade the application performance with fault tolerance support. Modern high-performance storage

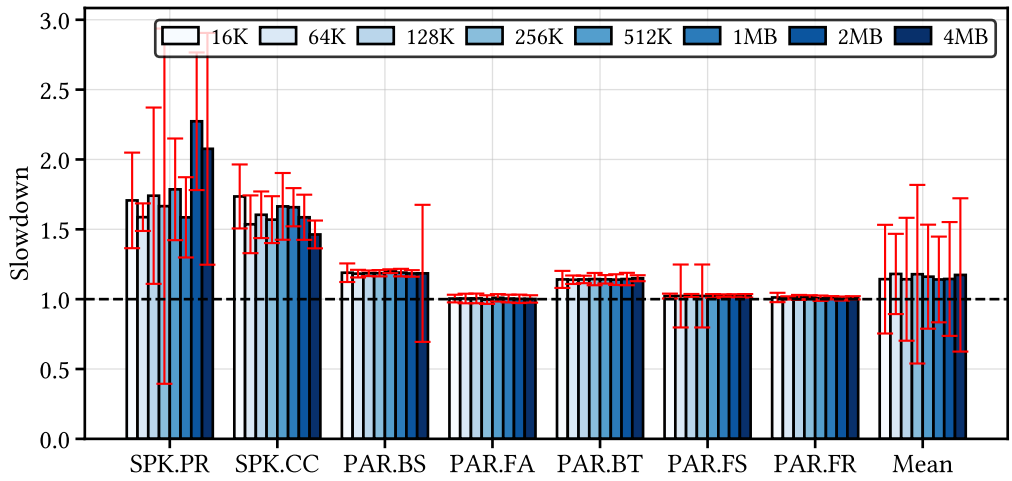


Figure 8.6 Effect of slab size on the execution time.

such Intel Optance SSD and SAMSUNG Z-SSD meets the performance requirements, which makes deploying RackMem promising for practical use.

8.3.4 RackDVS slab size and performance

RackMem allocates remote memory at slab granularity. A larger slab size reduces the total number of RPCs for allocation and reduces the total metadata footprint. On the other hand, a smaller slab size can provide better space utilization by reducing the unused space in slabs; however, the frequent slab allocation can be a bottleneck of the pagefault handler.

Figure 8.6 shows the normalized execution time of Spark and Parsec applications under intensive paging scenarios. We limit the available local memory for each workload by 40% of the total RSS size. The evaluation is repeated for different slab sizes from 16KB to 4MB.

The result shows little performance impact from using different slab sizes. RackMem effectively reduces the overhead of using smaller slab size with the two optimizations. First, RackMem maintains a small pool of slabs and keeps the pool populated. Thus, the local pool handles most of the slab requests and eliminates the RPC

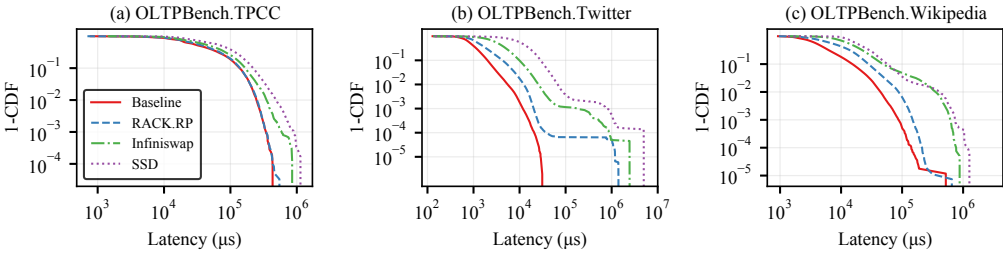


Figure 8.7 Distribution of transaction latency of OLTP-Bench (log scale, lower is better).

overhead in the critical path. Second, we implemented an RPC for the bulk allocation of slabs. The RPC allows RackMem can allocate multiple slabs on a remote node with one RPC. The two optimizations make RackMem tolerates the overhead of smaller slab sizes.

8.3.5 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 8.7 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 8.7. 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

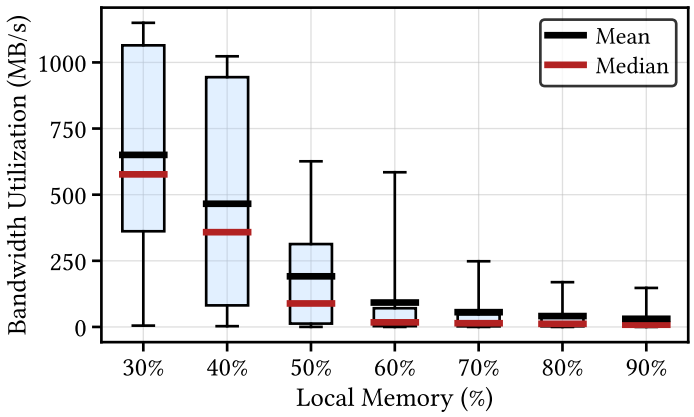


Figure 8.8 Network bandwidth utilization of applications with RackMem.

5.5ms, 12.4ms, 25.7ms, and 67.3ms, respectively.

8.3.6 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 5.5).

Figure 8.8 plots the network bandwidth utilization of RackMem with prefetch and reclaim activated (`RACK.RP`) for all applications from Figure 8.2. 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) [8], network bandwidth only gets saturated when 20 parallel applications running at a 30% local memory limit si-

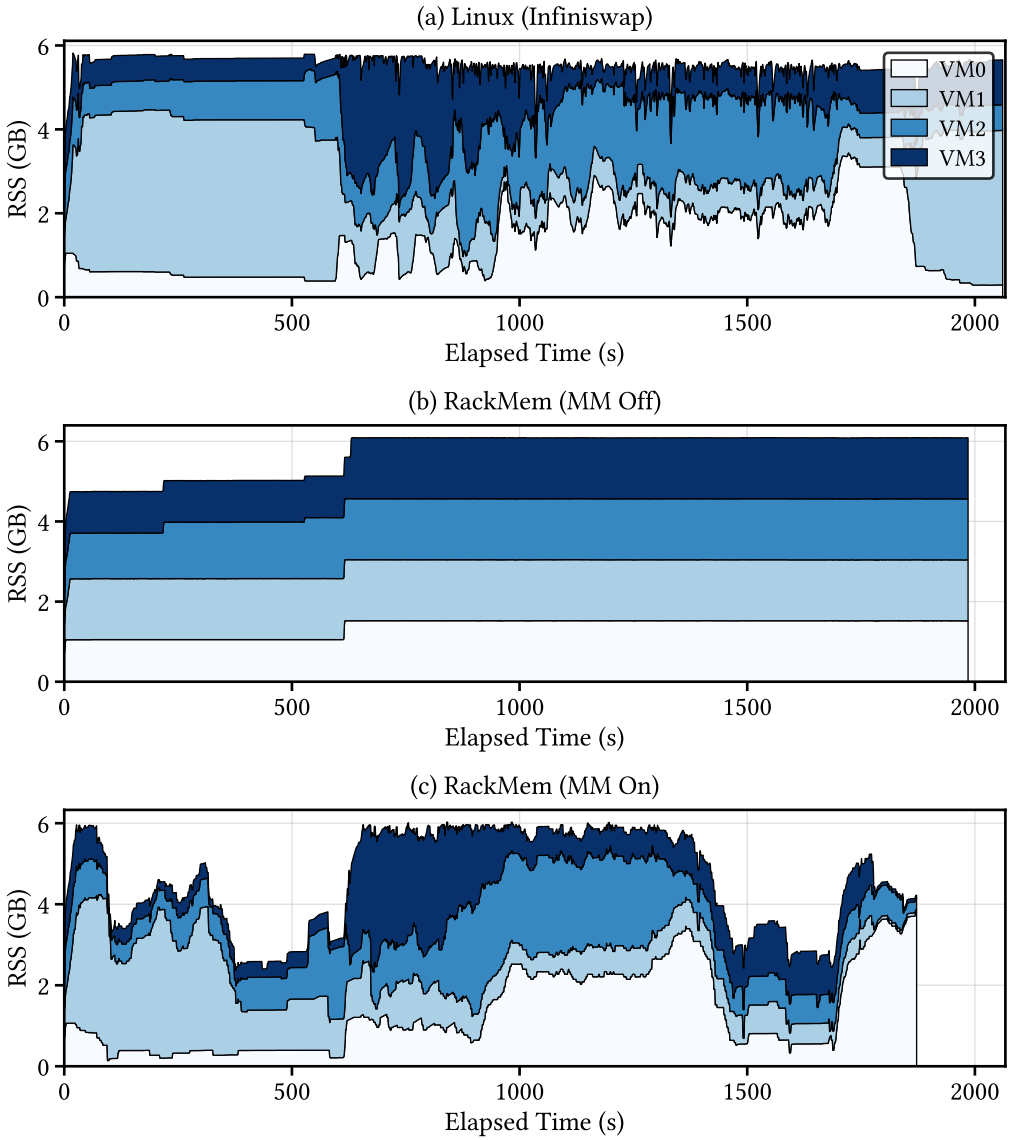


Figure 8.9 Dynamic local memory partitioning.

multaneously 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 4.4.1); 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 8.3.1 showed that RackMem achieves close to optimal performance at a 50% local memory ratio even without automatic re-balancing.

8.3.7 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 4.4.1) 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 8.9 plots the results. Figure 8.9 (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 8.9 (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

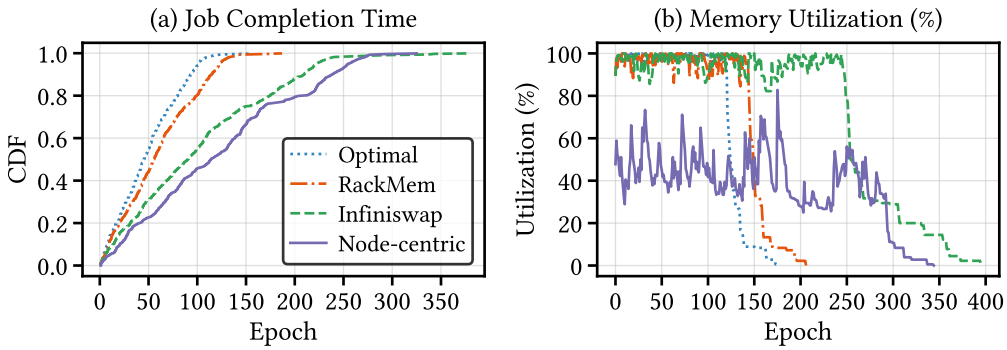


Figure 8.10 Rack-scale simulation of a job processing scenario.

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 8.9 (c) shows the result for Infiniswap with a total time to completion of 2,062s. Linux also shows a noisier pattern than RackMem, which results in more overhead for frequent repartitioning. The mean job completion times of RackMem with/without dynamic repartitioning and Infiniswap are 354s, 362s, and 400s, respectively.

8.3.8 Rack-scale Job Processing Simulation

Finally, we analyze the potential performance benefit of RackMem for job processing in a cluster with a simulation. Figure 8.10 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 8.3) 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 [28]. 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 8.2).

The cumulative distribution function (CDF) of the job completion time for the four scenarios is shown in Figure 8.10 (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 8.10 (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.

Chapter 9

Evaluation - Instant VM Live Migration

This section presents evaluation of Instant for YCSB benchmark [20] and comparison to other migration techniques. Our evaluation answers the following questions.

- How fast Instant can complete VM migration?
- How quickly Instant recovers from the performance degradation caused by the cold miss on the target node.

9.1 Experimental setup

The source and destination nodes are equipped with an Intel(R) Xeon(R) Silver 4114 2.20GHz processor (10-core, 20-thread) and 64 GiB of local memory. Each machine runs Ubuntu 18.04 with Linux kernel version 5.3.7. The machines are connected by a ConnectX-4 56g Infiniband NIC with a theoretical bandwidth of 56 Gbit/s and a latency of 0.6μs. The disaggregated memory and VM live migration traffic share the same Infiniband connection and can interfere. The VM network is physically separated and uses a 1 Gbit/s Ethernet NIC. QEMU/KVM version 4.2 provides the hypervi-

sor, and the VMs run Ubuntu 18.04 as their guest OS. The VMs use network-attached storage to access their disk images. VM disk I/O traffic is routed through the VM network. All VMs used in the experiments are equipped with 4-VCPU and 32 GB of local memory.

9.2 Target Applications

We evaluate Instant for an in-memory key-value database with the workloads in Yahoo Cloud Serving Benchmark (YCSB). We run a Redis server inside a VM and send requests from a node in the same cluster. We evaluate Instant for the following six workloads in YCSB.

- **Update Heavy (UH):** 50/50 mixed reads and writes.
- **Read Mostly (RM):** 95/5 mixed reads and writes.
- **Read Only (RO):** 100% reads requests.
- **Read Latest (RL):** The client inserts new records and sends read requests mostly on the recently added keys.
- **Short Ranges (SR):** Send requests on short range keys.
- **Read-Modify-Write (RMW):** The client read a key from the DB, modify, and write back the modified value.

9.3 Comparison targets

We compare Instant for the VM live migration techniques available in QEMU v4.2. Specifically, we use the following techniques for the comparisons.

- **`PRE`** uses iterative pre-copy to reduce downtime of VM live migration. We enable `MULTIFD` to increase the migration bandwidth on the fast network in our cluster.
- **`POST`** sends only minimal VM states and resumes the VM on the destination. The remaining pages on the source machine are fetched by the background task or pagefault handler.
- **`INSTANT`** is the stop-and-copy only version of Instant.

9.4 Database and client setups

We prepare two Redis databases that are having 1 million and 5 million keys separately. The RSS sizes of each VM are 3GB and 11GB separately. We refer to each database as **`1M`** and **`5M`**.

We stress the database with YCSB clients configured to use 4-thread or 20-thread. We refer to each configuration as **`4T`** and **`20T`** separately. For example, the scenario running a 20-thread client for a database having 5 million keys is **`5M/20T`**.

9.5 Memory disaggregation scenarios

We consider two memory disaggregation scenarios for the evaluation. The first scenario is allocating all pages in the local memory. Despite the ability to use remote memory, we consider it as a common scenario of Instant. Cluster scheduler places VM on the node having enough resources; it is a reasonable assumption that most VMs rarely experience memory pressure in datacenters. In this case, the VM migrated with `INSTANT` experiences burst remote paging after the migration.

The second scenario is migrating VM under memory pressure. It is also a common scenario of VM live migration for load balancing. In this case, part of the VM's memory is stored in the destination. The destination also has enough local memory to accommodate the target VM.

We deploy two nodes for the evaluation simulating the above scenarios.

9.5.1 Time-to-responsiveness

Time-to-responsiveness (TTR) is a useful metric to pinpoint the time of the VM performance is restored. For example, 10s for TTR (90%) means the performance does not degrade below 90% after 10s. We use a slightly modified definition of TTR, originally proposed by Irene et al. [68]. To compute TTR, we first normalize the YCSB latency by the mean latency without the migration overhead. The normalized latency becomes 1.0 if the observed latency is greater or equal to the mean latency. We compute the moving average of the normalized latency with a one-second window and draw the

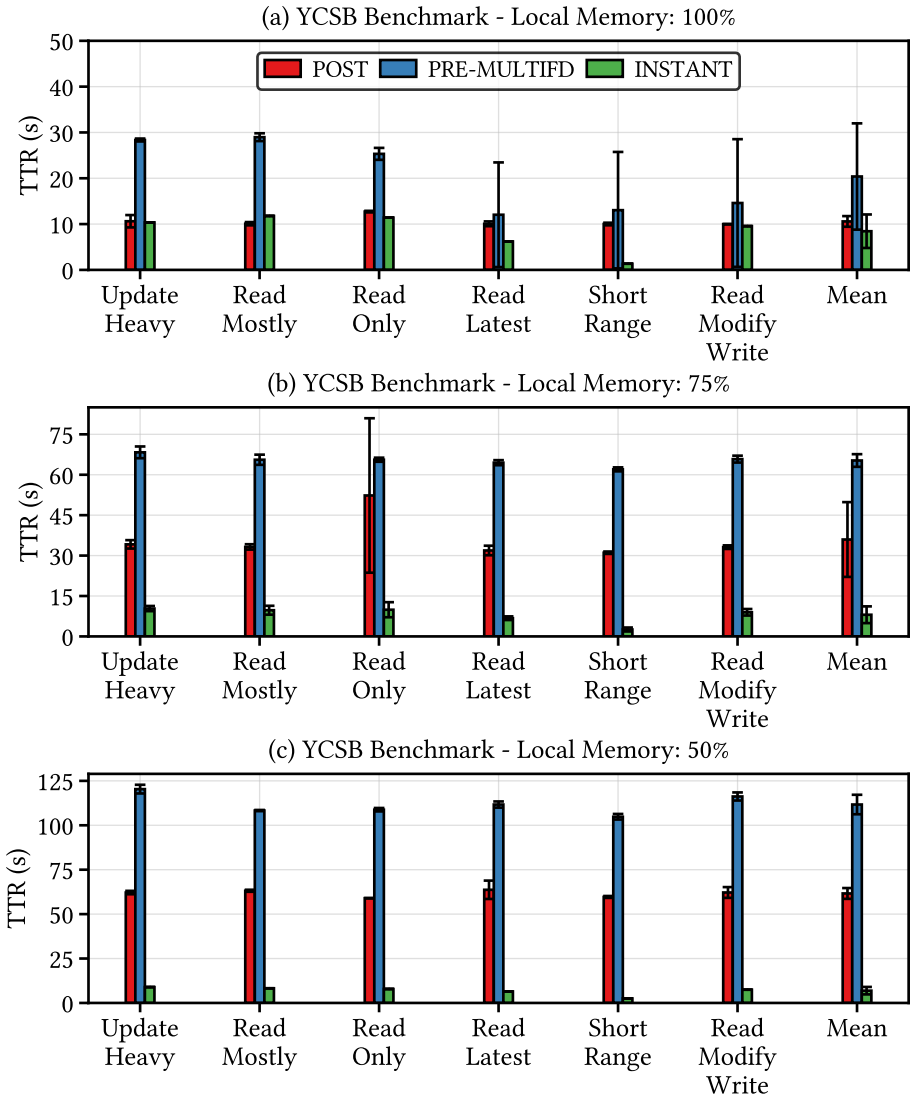


Figure 9.1 Time-to-responsiveness of live migration techniques.

TTR. We use 80% TTR for all scenarios as the performance metric.

Figure 9.1 shows the comparison of mean TTR for all migration techniques and YCSB workload configurations: `5M/4T`, `10M/4T`, and `5M/4T`. Figure 9.1 (a) is the result of migrating VMs without memory pressure. In this scenario, all local pages are

fetched on the destination in Instant. Instant does not get a performance benefit from the pre-existing pages on the destination; this scenario is the most unfavorable scenario for Instant. On average, POST, PRE, and Instant achieve 10.6s, 20.4s, and 8.4s TTRs separately. Instant reduces the TTRs by 20.1% and 59.0% over POST and PRE separately.

Figure 9.1 (b) migrates VMs under a light memory pressure; it stores 75% of the total memory footprint in the local memory and stores rest of them in remote memory (destination node in this scenario). In this scenario, a smaller subset of the pages are already available in the destination, Instant copies less pages in the migration phase than PRE and POST in this sceanrio. On average, POST, PRE, and Instant achieves 36.0s, 65.3s, and 8.0s TTRs for separately. Instant reduces the TTRs by 78.8% and 87.7% over POST and PRE separately. The performance improvement with Instant is much more significant due to other techniques start to suffer from the thrashing between the remote paging and the migration process.

Figure 9.1 (c) migrates VMs under a severe memory pressure. Only 50% of the pages compared to the total memory footprint are stored in the local memory. PRE and POST suffers from a significant paging overhead due to the thrashing. On average, POST, PRE, and Instant achieves 61.6s, 111.7s, and 6.9s TTRs for separately. Instant reduces the TTRs by 88.7% and 93.8% over POST and PRE separately.

Instant is the only technique that is constantly showing the single digit TTR in all configurations. Actually, the TTR is improved as the VM gets more memory pressure which shows the an unique characteristic of Instant. This result shows the *seamless* aspect of Instant technique.

9.5.2 Effective Downtime

Figure 9.2 shows *effective* downtime of each technique for the six YCSB workloads under different memory pressure scenarios. Effective downtime is not same to the reported downtime in QEMU. The reported downtime in QEMU only measures the latency of sending a small amount of data that includes processor states, device buffers, and VM metadata; the actual service interruption time can be highly differ from the reported value especially for the post-copy based techniques.

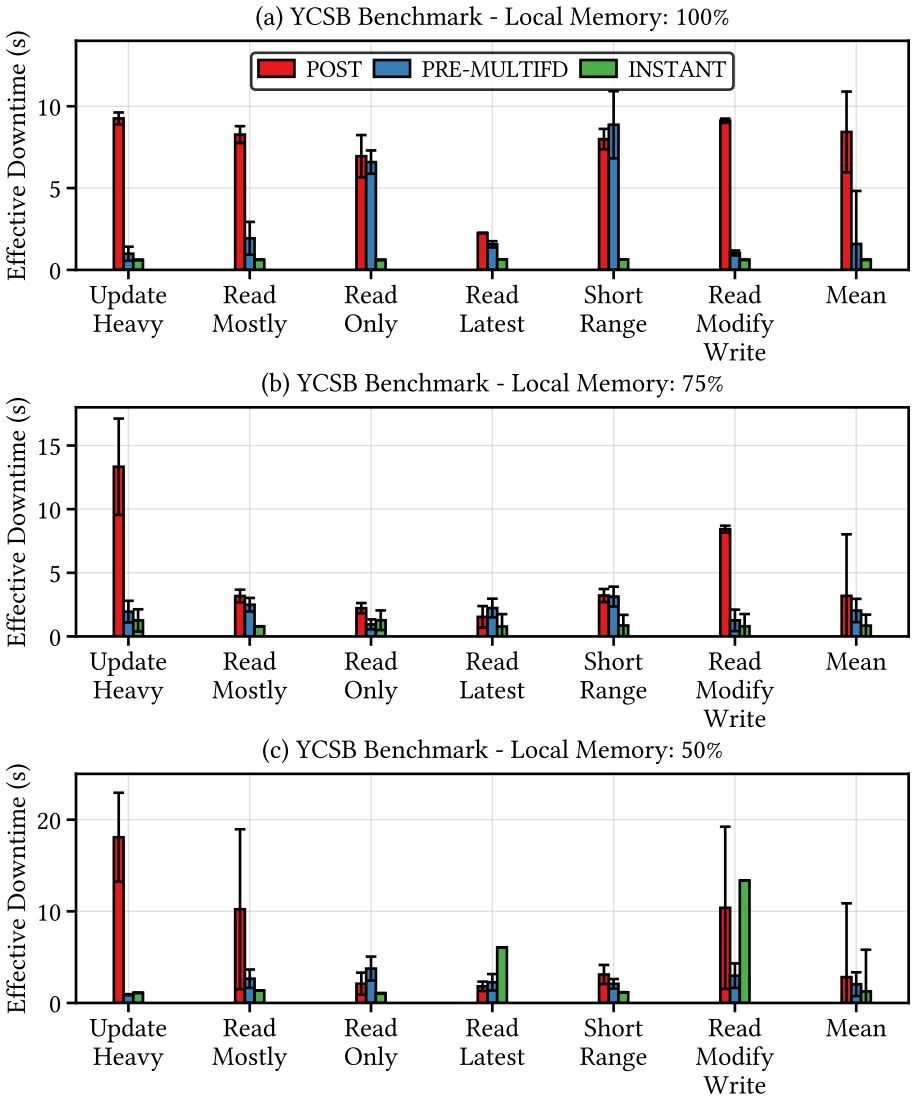


Figure 9.2 Effective downtime of live migration techniques.

The effective downtime measures the service interruption time *outside* of the VM. We define the effective downtime as the 99.95^{th} interval of two consecutive YCSB responses after the VM live migration is initiated.

Figure 9.2 (a) is the effective downtime from the VM live migrations without mem-

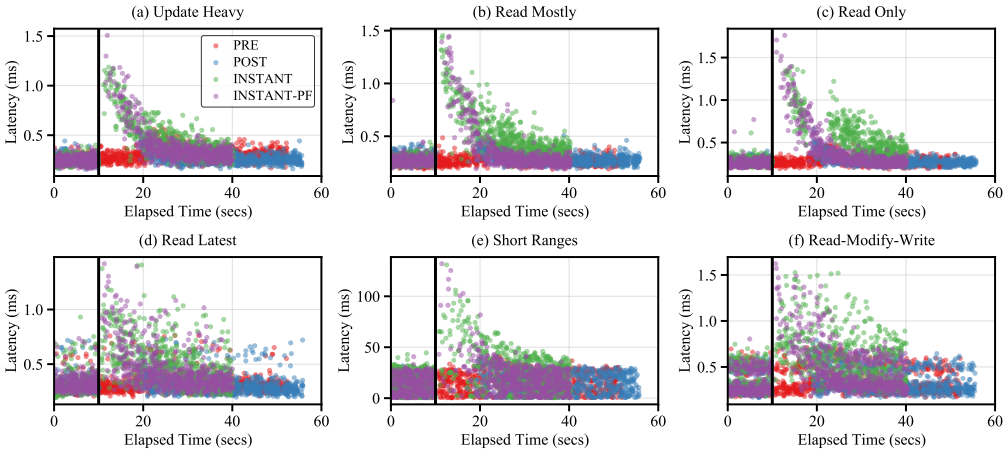


Figure 9.3 YCSB latency scatter plot (5M keys, 20-thread)

ory pressure. Effective downtime of `POST`, `PRE`, and INSTANT are 8.43s, 1.58s, and 0.62s for separately. INSTANT achieves the shortest effective downtime among the three techniques. INSTANT reduces the effective downtime by 92.6% and 60.7% over `POST` and `PRE` separately.

We can see the similar trends for the different memory pressure scenarios: Figure 9.2 (b) and (c). INSTANT is the only technique that achieves subsecond effective downtime. For the 75% and 50% local memory sceanrios, INSTANT achieves 0.61s and 0.84s of effective downtime separately.

9.5.3 Effect of INSTANT optimizations

Figure 9.3 shows a latency scatter plot of six YCSB workloads for `PRE`, `POST`, and INSTANT optimizations. We selectively show the `5M/20T` scenario that shows the difference of target techniques. We only include 1,000 random samples for each technique to draw the scatter plots with a reasonable number of data points. The latencies longer than the 99.95^{th} value are excluded to scale down Y-axis within a reasonable range, mostly the first few points at the beginning of the restoration phase are excluded.

`POST` does not show any point in the early stage of restoration. It starts to show points after 20 seconds after the beginning. `INSTANT` does not run background fetch

task in the destination. `INSTANT` shows a slightly better latency at the beginning, but the full restoration of the performance is much slower than `INSTANT-PF` which is running a background task to fetch working set pages from the source machine.

Chapter 10

Conclusion

Virtualized environment is a promising target for applying remote memory. Virtual machines run in isolated environments, and seamless remote memory access is easily enabled through hypervisor. The remote memory enabled VMs can tolerate under memory pressure by utilizing remote to store infrequently accessed data. As a demonstration of the idea, we presented RackMem. RackMem enables seamless VM execution on remote memory by minimizing software overhead with specialized remote paging stacks for modern high-performance networking devices. The set of optimizations in RackMem eliminates the page reclamation latency in the critical path and reduces the number of page faults. In the rack-scale simulation, RackMem enabled cluster significantly improved the mean job processing time (40.9%).

In addition to RackMem, we present the instant and seamless migration of virtual machines. We observed the remote memory unexpectedly solves the long-standing problem in VM live migration (i.e. entire memory copy). The ability of location-independent memory access and efficient remote paging mechanism enables instant migration of virtual machines by only sending minimal metadata. The efficient remote paging mechanism in RackMem also provides significantly better restoration performance than the post-copy implementation in QEMU/KVM. We demonstrate

the performance improvement of instant VM live migration with in-memory key-value workloads in YCSB benchmark.

RackMem and instant VM live migration solve the fundamental limitations in the traditional datacenter architecture that limiting resource efficiency. By applying RackMem, data center jobs can be placed onto more nodes and improve the overall utilization. In addition to that, the instant migration mechanism quickly resolves the load imbalance in the data centers by quickly moving VMs to less loaded servers. We expect the future data centers will actively deploy similar approaches to push the limit of the current resource efficiency.

10.1 Future Directions

The proposed techniques in this dissertation open new interesting future research directions.

Warehouse-scale memory management. Allocating the right amount of local memory for VMs is important to maximize the utility of the limited resources. In RackMem, we presented a local memory manager for intra-node VMs. Scaling up the local memory manager for warehouse-scale will be interesting and important for the datacenter-wide resource efficiency. A more sophisticated algorithm could be used to find the right allocation of local memory, such as miss ratio curve prediction [66, 69] and machine learning. An efficient algorithm for finding a global VM placement and resource allocation will also be interesting future work.

Pre-fetch with memory access prediction. Memory access with pagefault handling is still a few orders of magnitude slower than local memory access (10000ns vs. 60ns). Prefetching with accurate memory access prediction can potentially eliminate the page fault handling overhead in remote paging systems. With ideal systems, the VM under intensive memory pressure will run more seamlessly.

We expect machine learning approach for memory access prediction would be promising [16, 30]. Applying such techniques in the kernel space would be an interesting research direction since kernel space has little time budget for the prediction.

Frequent VM live migration for idle resource utilization. Instant VM live

migration reduces the total migration time of VM order of magnitude faster than the traditional approaches. Only our approach can migrate a VM back and forth among multiple servers in milliseconds scale by avoiding the entire memory copy. This ability enables utilizing a short period of resource variability existing in millions of VMs in data centers that have not been utilized due to the lack of a proper mechanism. For example, if there are periodic patterns of ephemeral idle resources (e.g., 10s), we can migrate a VM under resource pressure to utilize the short period of idle resources. We expect there are plenty of rooms for improving resource utilization in a shorter time scale. Applying instant VM live migration will be a promising approach for addressing this issue.

Bibliography

[1] About gen-z | gen-z consortium | open-systems interconnect. https://genzconsortium.org/about-us/. (Accessed on 05/24/2021).

[2] Ccix. https://www.ccixconsortium.com/. (Accessed on 05/24/2021).

[3] Connectx-7 ethernet datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/networking/ethernet-adapters/connectx-7-datasheet-Final.pdf. (Accessed on 05/24/2021).

[4] Gartner retires the magic quadrant for x86 server virtualization infrastructure. https://www.gartner.com/en/documents/3642418/gartner-retires-the-magic-quadrant-for-x86-server-virtua. (Accessed on 05/24/2021).

[5] Home - xen project. https://xenproject.org/. (Accessed on 05/24/2021).

[6] Intel confirms retreat on omni-path. https://www.hpcwire.com/2019/08/01/report-intel-retreats-on-omni-path/. (Accessed on 05/24/2021).

[7] The machine: A new kind of computer. https://www.hpl.hp.com/research/systems-research/themachine/. (Accessed on 05/24/2021).

[8] Nvidia mellanox connectx-6 vpi adapter cards | nvidia. https://www.nvidia.com/en-us/networking/infiniband-adapters/connectx-6/. (Accessed on 05/24/2021).

[9] Opencapi consortium: Official site. https://opencapi.org/. (Accessed on 05/24/2021).

[10] rdma-core/ibverbs.h at master · linux-rdma/rdma-core. https://github.com/linux-rdma/rdma-core/blob/master/libibverbs/ibverbs.h. (Accessed on 08/10/2021).

[11] Y. Abe, R. Geambasu, K. Joshi, and M. Satyanarayanan. Urgent virtual machine eviction with enlightened post-copy. In *Proceedings of The12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '16, page 51–64, New York, NY, USA, 2016. Association for Computing Machinery.

[12] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, S. Novaković, A. Ramanathan, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 775–787. USENIX Association, 2018.

[13] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote memory in the age of fast networks. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 121–127. ACM, 2017.

[14] 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 *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[15] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: shared memory computing on networks of workstations. *Computer*, 29(2):18–28, 1996.

[16] G. Ayers, H. Litz, C. Kozyrakis, and P. Ranganathan. Classifying memory access patterns for prefetching. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 513–526, New York, NY, USA, 2020. Association for Computing Machinery.

[17] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, et al. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[18] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[19] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design and Implementation - Volume 2*, NSDI'05, 2005.

[20] 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.

[21] J. Corbet. Fun with tracepoints [lwn.net]. https://lwn.net/Articles/346470/, 8 2009. (Accessed on 05/24/2021).

[22] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proc. VLDB Endow.*, 7(4):277–288, Dec. 2013.

[23] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, Apr. 2014. USENIX Association.

[24] L. Foundation. mm, swap: use rbtree for swap extent. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=4efaceb1c5f8136d5fec3f26549d294b8e898bd7, 7 2019.

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

[26] L. Foundation. Null block device driver. https://www.kernel.org/doc/html/latest/block/null_blk.html, 2020.

[27] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network requirements for resource disaggregation. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16, pages 249–264. USENIX Association, 2016.

[28] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667. USENIX Association, 2017.

[29] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, and Y. Bao. Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces. In *Proceedings of the International Symposium on Quality of Service*, IWQoS '19, New York, NY, USA, 2019. Association for Computing Machinery.

[30] M. Hashemi, K. Swersky, J. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan. Learning memory access patterns. In J. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1919–1928. PMLR, 10–15 Jul 2018.

[31] M. R. Hines and K. Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, New York, NY, USA, 2009. Association for Computing Machinery.

[32] Infiniswap: Efficient memory disaggregation with infiniswap. https://github.com/SymbioticLab/infiniswap, 2017.

[33] H. Jin, L. Deng, S. Wu, X. Shi, and X. Pan. Live virtual machine migration with adaptive, memory compression. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10, Aug 2009.

[34] C. Jo and B. Egger. Optimizing live migration for virtual desktop clouds. In *IEEE 5th International Conference on Cloud Computing Technology and Science*, volume 1 of *CloudCom '13*, pages 104–111, Dec 2013.

[35] C. Jo, E. Gustafsson, J. Son, and B. Egger. Efficient live migration of virtual machines using shared storage. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '13, pages 41–50, New York, NY, USA, 2013. ACM.

[36] C. Jo, H. Kim, H. Geng, and B. Egger. Rackmem: A tailored caching layer for rack scale computing. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, PACT '20, page 467–480, New York, NY, USA, 2020. Association for Computing Machinery.

[37] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter rpcs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, Boston, MA, Feb. 2019. USENIX Association.

[38] A. Kalia, M. Kaminsky, and D. G. Andersen. Using rdma efficiently for key-value services. *SIGCOMM Comput. Commun. Rev.*, 44(4):295–306, Aug. 2014.

[39] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 437–450, Denver, CO, June 2016. USENIX Association.

[40] A. Kalia, M. Kaminsky, and D. G. Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 185–201, Savannah, GA, Nov. 2016. USENIX Association.

[41] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee. Snucl: An opencl framework for heterogeneous cpu/gpu clusters. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, page 341–352, New York, NY, USA, 2012. Association for Computing Machinery.

[42] K. Koh, K. Kim, S. Jeon, and J. Huh. Disaggregated cloud memory with elastic block management. *IEEE Transactions on Computers*, 68(1), 2018.

[43] 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 *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 317–330, New York, NY, USA, 2019. ACM.

[44] 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 *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, page 267–278, New York, NY, USA, 2009. Association for Computing Machinery.

[45] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level implications of disaggregated memory. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12, Feb 2012.

[46] Z. Liu, W. Qu, W. Liu, and K. Li. Xen live migration with slowdown scheduling algorithm. In *Proceedings of the 2010 International Conference on Parallel and Distributed Computing, Applications and Technologies*, PDCAT '10, pages 215–221, Washington, DC, USA, 2010. IEEE Computer Society.

[47] Y. Lu, J. Shu, Y. Chen, and T. Li. Octopus: an rdma-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 773–785. USENIX Association, 2017.

[48] H. A. Maruf and M. Chowdhury. Effectively prefetching remote memory with leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 843–857. USENIX Association, July 2020.

[49] Mellanox. Mellanox products: Connectx®-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, 6 2016.

[50] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, Oct 2001.

[51] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-tolerant software distributed shared memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 291–305, Santa Clara, CA, July 2015. USENIX Association.

[52] M. Nelson, B.-H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*, Anaheim, CA, Apr. 2005. USENIX Association.

[53] T. Newhall, S. Finney, K. Ganchev, and M. Spiegel. Nswap: A network swapping module for linux clusters. In *Euro-Par 2003 Parallel Processing*, pages 1160–1169, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[54] S. Novakovic, Y. Shan, A. Kolli, M. Cui, Y. Zhang, H. Eran, B. Pismenny, L. Liss, M. Wei, D. Tsafrir, and M. Aguilera. Storm: A fast transactional dataplane for remote data structures. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, page 97–108, New York, NY, USA, 2019. Association for Computing Machinery.

[55] pmem.io. Persistent memory programming. http://pmem.io/, 8 2020.

[56] QEMU. Qemu. https://www.qemu.org/. (online, accessed July 2020).

[57] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, New York, NY, USA, 2012. Association for Computing Machinery.

[58] A. Ruprecht, D. Jones, D. Shiraev, G. Harmon, M. Spivak, M. Krebs, M. Baker-Harvey, and T. Sanderson. Vm live migration at scale. In *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '18, New York, NY, USA, 2018. Association for Computing Machinery.

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

[60] Y. Shan, S.-Y. Tsai, and Y. Zhang. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 323–337. ACM, 2017.

[61] A. Shribman and B. Hudzia. Pre-copy and post-copy vm live migration for memory intensive applications. In *European Conference on Parallel Processing*, pages 539–547. Springer, 2012.

[62] P. Stuedi, A. Trivedi, J. Pfefferle, A. Klimovic, A. Schuepbach, and B. Metzler. Unification of temporary storage in the nodekernel architecture. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 767–782, Renton, WA, July 2019. USENIX Association.

[63] P. Svärd, B. Hudzia, J. Tordsson, and E. Elmroth. Evaluation of delta compression techniques for efficient live migration of large virtual machines. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '11, pages 111–120, New York, NY, USA, 2011. ACM.

[64] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes. Borg: The next generation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[65] S.-Y. Tsai and Y. Zhang. Lite kernel rdma support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 306–324. ACM, 2017.

[66] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad. Efficient MRC construction with SHARDS. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 95–110, Santa Clara, CA, Feb. 2015. USENIX Association.

[67] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud '10, pages 10–10. USENIX Association, 2010.

[68] I. Zhang, A. Garthwaite, Y. Baskakov, and K. C. Barr. Fast restore of checkpointed memory using working set estimation. In *Proceedings of the 7th ACM*

SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '11, page 87–98, New York, NY, USA, 2011. Association for Computing Machinery.

[69] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, page 177–188, New York, NY, USA, 2004. Association for Computing Machinery.

요약

인공지능 및 빅데이터 컴퓨팅의 유행으로 인해 클라우드 환경으로의 이전이 가속되고 있다. 클라우드 환경은 거대한 연산 자원을 상시 가동할 필요 없이 원하는 순간 원하는 양의 대한 연산 비용만을 지불하면 되기 때문에, 고객의 경우 서버 유지에 대한 비용을 크게 절감할 수 있고 클라우드 서비스를 제공하는 데이터센터는 연산 자원의 이용 효율을 극대화 할 수 있다.

이러한 시나리오에서 데이터센터 입장에서는 연산 자원 활용 효율을 개선하는 것이 중요한 목표가 된다. 최근 데이터센터의 규모는 사상 유래없는 규모로 증가하고 있기 때문에 작은 효율 개선으로도 막대한 경제적 가치를 창출 할 수 있다.

데이터 센터의 효율은 위치 선정, 구조 설계, 냉각 시스템, 하드웨어 구성 등등 다양한 요소들에 영향을 받지만, 이 논문에서는 특히 연산 및 메모리 자원을 관리하는 소프트웨어 설계 및 구현을 다룬다.

이 학위 논문에서는 데이터 센터 효율 개선을 위한 소프트웨어 기반 기술을 제안하며 크게 두가지 요소의 큰 기여를 만든다. 첫째로 우리는 가상화 환경을 위한 소프트웨어 기반 메모리 분리 시스템을 제안한다. 최근 고속 네트워크의 발전으로 인해 원격 메모리 접근 비용이 획기적으로 줄어 들었고, 이 논문에서는 현대 네트워킹 하드웨어를 기반으로 원격 메모리 위에서 실행되는 가상 머신의 큰 성능 저하 없이 실행할 수 있음을 보인다. 제안된 기술이 적용된 가상머신 하이퍼바이저 QEMU/KVM 를 통한 실험 결과, 본 논문에서 제안한 기법은 기존 시스템 대비 원격 페이징에 대한 꼬리 지연시간을 98.2% 개선한다. 또한 랙 규모의 작업처리 시뮬레이션을 통한 실험에서, 제안된 시스템은 전체 작업 처리 시간을 기존 시스템 대비 40.9% 개선한다.

두 번째 기여는, 원격 메모리를 이용하는 즉각적인 가상머신 이주 기법이다. 가상화 환경의 원격 메모리 활용에 대한 확장은 그것만으로 자원 이용률 향상에 대해 큰 기여를 하지만, 여전히 한 서버에서 여러 어플리케이션이 경쟁적으로 자원을 이용하는 경우 성능이 크게 저하 될 수 있다. 이 논문에서 제안하는 즉각적인 가상머신 이주 기법은 원격 메모리 상에서 아주 작은 메타데이터의 전송만으로 가상머신의 이주를 가능하게 한다. 메모리 상에 키와 값을 저장하는 데이터베이스 벤치마크를 실행하

는 가상머신을 기반으로 한 평가에서, 제안된 가상머신 이주 기법은 기존 기법대비 실질적인 서비스 중단시간을 최대 92.6% 개선한다.

이 학위 논문에서는 제안된 원격 메모리 시스템에 대한 두가지 소프트웨어 기법을 데이터센터에 적용할 경우 획기적인 연산 및 메모리 자원 활용 효율 개선을 가져올 것으로 기대된다.

주요어: 가상화, 실시간 이주, 원격 메모리, 원격 직접 메모리 접근
학번: 2012-20869