Hierarchical Orchestration of Disaggregated Memory

Wenqi Cao and Ling Liu, *Fellow, IEEE*

Abstract—This article presents *XMemPod*, a hierarchical disaggregated memory orchestration system. *XMemPod* virtualizes cluster wide memory to scale large memory workloads in virtualized clouds. It makes three novel contributions: (1) *XMemPod* offers efficient, transparent, and dynamic sharing of available memory that is disaggregated across VMs on the same host or in the cluster. (2) *XMemPod* provides a hierarchical memory expansion framework, which enables memory-intensive workloads on a VM to expand its memory demand over virtualized host memory first, and remote memory next, before resorting to external disk. (3) *XMemPod* provides a suite of optimization techniques to further improve the utilization and access latency of disaggregated memory. *XMemPod* is deployed on a virtualized RDMA cluster without any modifications to user applications and the OSes. Evaluated with multiple workloads on unmodified Spark, Apache Hadoop, Memcached, Redis and VoltDB, using *XMemPod*, throughputs of these applications improve by 11x to 612x over conventional Linux, and by 1.7x to 14x over the existing representative remote memory paging systems, and yet the total amount of network traffic consumed by *XMemPod* is only 24 percent of the existing approaches.

Index Terms—Memory disaggregation, virtualization, operating system

1 Introduction

Big data and latency-demanding applications [7], [8], [9], [53], [58] are typically deployed using the application deployment models, comprised of virtual machines (VMs), containers, and/or executors/JVMs. These applications enjoy high throughput and low latency if they are served entirely from memory. However, actual estimation and memory allocation are difficult. When these applications cannot fit their working sets in real memory of their VMs/containers/executors, they suffer large performance loss due to excess page faults and thrashing. Even when available (unused) memory is present in other VMs/containers/executors on the same host or a remote node, these applications are unable to share those unused host/remote memory (Section 2).

Existing research studied the above problems from two orthogonal dimensions: (i) estimating working set size for accurate resource allocation and (ii) increasing effective resource capacity for executors. Accurate memory allocation is hard as peak memory variations happen under different application types, workload inputs, data characteristics and traffic patterns. Applications often overestimate their requirements or attempt to allocate for peak usage [17], [27], resulting in severely unbalanced memory usage across executors, underutilization on the host and across the cluster [48]. Instead, proposals for increasing effective memory capacity promote the allocation of global memory resource shared by all machines (VMs/containers) to increase their effective memory capacities. These proposals promote new architectures and new hardware design for memory disaggregation [10], [22], or new programming models [42], [47]. But they lack of desired transparency at OS, network stack, or application level, hindering their practical applicability. Recent efforts represented by Accelio nbdX [1], including Infiniswap [24] built on top of nbdX, exploit RDMA networks for remote memory paging with transparency. However, they solely rely on RDMA pre-registered memory allocation for remote memory sharing and cannot extend RDMA for sharing available physical memory on the same node across VMs or containers.

This paper describes *XMemPod*, a hierarchical disaggregated memory orchestration system, which accelerates big-data and machine learning (ML) workloads by virtualizing two types of cluster-wide external memory: (1) The available memory from other VMs on the same node due to inter-VM memory usage imbalance, which provides memory-speed access; and (2) The available remote memory in the cluster through RDMA registered memory allocation, which provides network-speed access. *XMemPod* is designed to dynamically orchestrate terabytes of cluster wide memory into multiple memory pods of these two types. By leveraging guest indirection, *XMemPod* transparently virtualizes cluster-wide available memory into the physical address space of each guest OS instance on demand (Section 3).

XMemPod is inspired by existing proposals from two orthogonal efforts (Section 6): exploiting non-intrusive sharing of memory between host and its guest VMs [30], [56], [57] and exploiting the disk-network latency gap via available remote memory [29], [60]. However, unlike existing proposals for dynamic memory balancing, *XMemPod* does not pay any CPU overhead and time delay for memory scanning to detect memory imbalance and does not rely on

• *The authors are with the School of Computer Science, Georgia Institute of Technology, Atlanta, GA 30332. E-mail: {wcao39, lingliu}@cc.gatech.edu.*

Manuscript received 26 Apr. 2019; revised 8 Jan. 2020; accepted 15 Jan. 2020. Date of publication 21 Jan. 2020; date of current version 8 May 2020. (Corresponding author: Wenqi Cao.) Recommended for acceptance by J. Shalf. Digital Object Identifier no. 10.1109/TC.2020.2968525

estimation of the working set size of each VM/container/executor to respond to memory pressure, since accurate estimation of working set size is difficult under changing workloads [11], [16]. Moreover, existing solutions for remote memory paging [1], [24] solely rely on RDMA memory registration for network speed memory sharing with remote node (s), and cannot be used to leverage available memory in other VMs on the same node for memory speed sharing across VMs on the same host.

XMemPod addresses the above challenges by making the two types of disaggregated memory pods available to unmodified VMs for use via a low-latency in-memory block device. These memory pods serve transient overflow data or memory paging events for the VM that requires larger physical memory than its allocation. *XMemPod* does so without any modifications to its applications or guest OS.

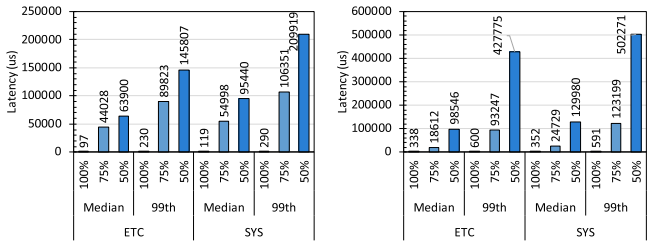
XMemPod meets the memory demand of a VM by prioritizing the use of available memory from other VMs on the same host via the memory-speed I/O device. Upon insufficient available memory on a local host, *XMemPod* resorts to remote memory pods via the RDMA port as the next tier network-speed I/O device. To scale the management cost of large size clusters, XMemPod reliably partitions the cluster-wide external memory into non-intrusive, elastic remote memory sharing groups through decentralized coordination.

XMemPod is implemented on Linux kernel 4.1.0 and KVM [35], deployed on a 56 Gbps, 32-machine RDMA cluster in Emulab [5] with 2 terabyte (TB) collective memory. We evaluated it using multiple unmodified memory intensive applications: Spark [58], [59], Apache Hadoop [53], Memcached [7], Redis [8], VoltDB [9], and open benchmarks: HiBench [31], SparkBench [37] and YCSB [20]. Using *XMemPod*, throughputs of these applications improve by 11x to 612x, median and tail latencies improve by 174x to 702x and 218x to 630x respectively, over conventional OS swap facility. Compared with the existing approaches such as nbdX or Infiniswap, using *XMemPod*, throughputs improve between 1.7x and 14x, median latencies and tail latencies improve between 1.8x and 12x, and 3.3x and 15x respectively (Section 5).

There are situations that *XMemPod* does not yet handle effectively, including the simple first-come-first-serve policy for using the host coordinated shared memory on each node, which could be improved by using a more sophisticated policy, such as threshold-caped. Also we were unable to obtain larger RDMA clusters to perform stress test on scalability.

2 Motivation

The Effect of Transient Memory Usage Variation. Memory utilization imbalance and temporal usage variations are frequently observed in virtualized clouds [17], [23], and production datacenters [24], [53]. One study on a google 12K-machine cluster running a mixture of long and short-lived workloads reported around 50 percent memory utilization, stating "the gap between resource requests and average usage accounted for most of this difference" [48], [49]. Another study on two production datacenters (3,000-machine Facebook analytic cluster and 12,500-machine google cluster) reports severe imbalance in memory utilization for more than 70 percent of the time across machines [24].



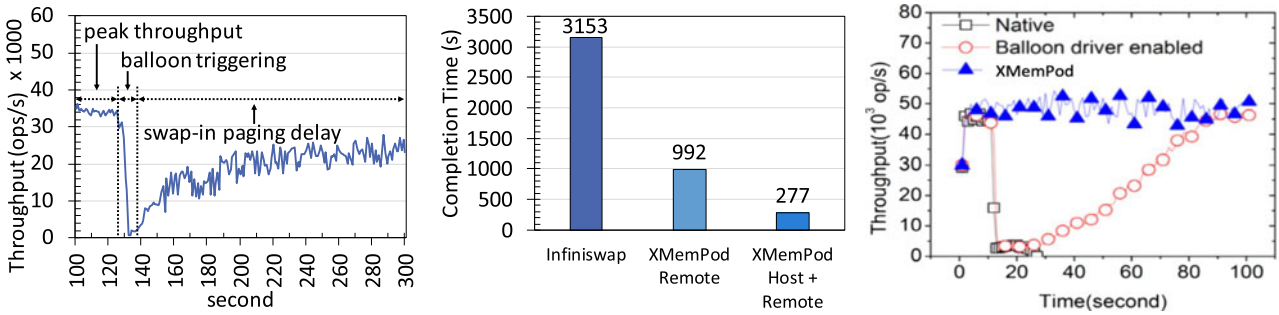
(a) Redis (b) VoltDB

Fig. 1. The effect of transient memory usage variation.

To gain an in-depth understanding of the potential opportunities of exploiting available memory of other VMs on the same node or in the cluster for performance speed-up of VMs under transient memory pressure, we run our micro benchmark on two bigdata applications: Redis [8], an in-memory NoSQL store, and VoltDB [9], a translytical in-memory database. We measure per server performance for each application with two benchmark workloads: ETC and SYS (for detailed setup see Section 5), using the raw dataset of 20 GB from YCSB [20] per server.

The peak memory that can fit the full working set is measured for both Redis and VoltDB, which are 29 GB and 30 GB respectively. We run each application on VM with three different memory capacity configurations – 100, 75, and 50. x% indicates that the VM is configured to run an application and can hold x% of its working set in memory. When $x = 100\%$, no paging occurs, but when $x < 100\%$, paging happens, as shown in Fig. 1. We highlight two observations. *First*, compared to 100 percent configuration, when 25 percent of the working set does not fit in memory (75 percent configuration), the median latency is worsened by 462x and 70x for Redia and VoltDB respectively. For 50 percent configuration (another 25 percent reduction), median latencies are worsened by 802x and 369x for Redis and VoltDB respectively. *Second*, For 75 percent configuration, 99th percentile latencies of Redis and VoltDB are worsened by 391x and 208x respectively, compared to 100 percent configuration. For 50 percent configuration, 99th percentile latencies are degraded by 724x and 850x for Redis and VoltDB respectively.

Dynamic Memory Balancing With the Balloon Driver. To cope with such harsh performance degradation when the working sets of applications do not fit in memory, memory ballooning [56] was proposed to move unused memory between VMs. However, solely relying on ballooning, applications under memory pressure still suffer hefty performance degradation due to three types of delays: the timing delay of scheduling ballooning, the balloon driver delay for moving sufficient memory, and the time delay for applications to return to their peak performance. Fig. 2a shows the three types of delays, the performance deterioration incurred, and the adverse effect on application performance, when the working set of Redis (ETC workload) does not fit in memory (50 percent configuration). We observed that the throughput of Redis deteriorates sharply during the 15 seconds from 123rd to 138th second, due to the excessive paging-out events, the timing delay of scheduling ballooning, and the balloon driver delay for moving sufficient memory. Upon installing sufficient ballooning memory at the 140th second, Redis throughput starts to recover but at a very slow pace. This is



(a) Ballooning Delays (b) Comparison of Solutions (c) XMemPod

Fig. 2. Alternative solution approaches.

because even with additional memory, Redis cannot immediately regain its peak performance due to the slow per page swap-in operation upon each page fault. It takes approximately 140 seconds to fully utilize the newly ballooned memory. Furthermore, ballooning incurs the overheads of scheduling and moving (deflating and inflating) memory from one VM to another on the same host, and cannot leverage the available remote memory in the cluster.

Differentiating Local versus Remote External Memory. In virtualized clouds, multiple VMs run on the same host machine, each having its own memory allocation (typically equal amount). When a VM reaches its memory limit, it has two alternative low-latency memory expansion opportunities before paging to local disk swap partition. They are available (unused) memory on local host and available (unused) remote memory in the cluster. Note that even though SSD is much faster than HDD, it is still two orders of magnitude slower than fast interconnect like InfiniBand based RDMA [54]. At the same time, RDMA interconnect is several times slower than memory-speed I/O on local node [45], making latency-sensitive prioritization critical for efficient disaggregated memory orchestration.

XMemPod implements a virtual block device manager, which provides a hierarchical disaggregated memory orchestration service to each VM by leveraging indirection, and interfacing with the host shared memory first, the remote memory next, and resorting only to the disk I/O for exception handling in the presence of insufficient remote memory. Fig. 2b shows the comparison of three alternative design choices by running Redis ETC workload under 50 percent configuration: (a) using existing solutions (e.g., Infiniswap) for remote memory sharing via RDMA, (b) *XMemPod* remote, which optimizes the remote RDMA sharing through bandwidth-aware batch swap-in and swap-out, and (c) hierarchical *XMemPod*, which leverages available memory on the host first and then available remote memory in the cluster, in addition to RDMA bandwidth optimizations (Section 4). Using hierarchical *XMemPod* in case (c), Redis completion time is 11.3x times faster than existing remote memory solutions in case (a), and 3.2x faster than *XMemPod* RDMA in case (b).

Fig. 2c shows the performance comparison with XMemPod turned on. We observe that with hierarchical orchestration of disaggregated memory, XMemPod can significantly improve the runtime performance of Redis when the working set of its data cannot fit fully into the main memory allocated at the configuration time. We would like to note that the comparison with existing systems is fair for two reasons. First, it is fair if the available host memory for XMemPod can be directly distributed to the VMs on the same host. Second, all solutions have the same amount of host memory. The main difference is that the conventional approach to VM management is to equally allocate the available host memory to all VMs on the host. In contrast, XMemPod takes back some proportion of the allocated VM memory to create a pool of host coordinated shared memory, say 1/10 or 1/3 of the 30 GB allocated memory to each VM on a host. XMemPod utilizes this pool of shared memory to compensate those VMs that are under transient memory pressure through providing host and remote memory paging service.

3 *XMemPod* Overview

XMemPod consists of four core components: the disaggregated memory paging service manager (PSM), the shared memory manager (SSM), the remote memory manager (RMM) and the disk swap partition manager (DSM), as shown in Fig. 3. They coordinate closely to provide efficient virtualization of cluster wide memory and hierarchical orchestration of disaggregated memory sharing capabilities. Client module and server module are the two main functional components of *XMemPod*. A VM that is low on memory will run the *XMemPod* client module to acquire external disaggregated memory service. Every node in the cluster can run the *XMemPod* server

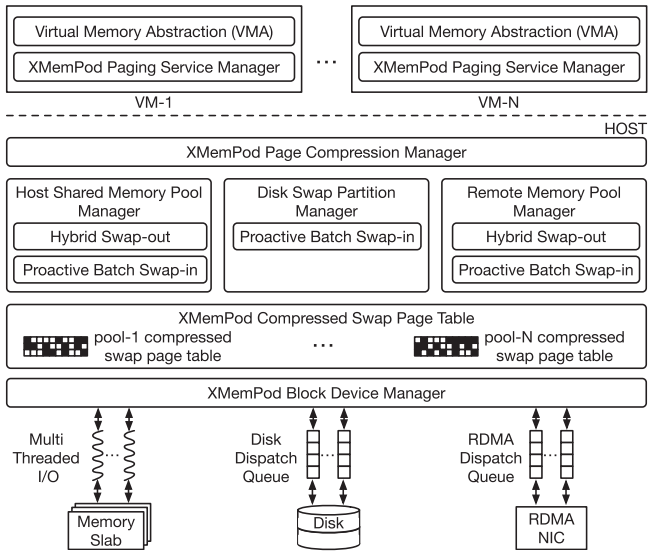


Fig. 3. *XMemPod* System architecture.

module to provide external memory expansion and sharing in response to the requests from *XMemPod* clients in a cluster. *XMemPod* is currently implemented on a VM based applications deployment platform, which is common in production datacenters with virtualized clouds [17], [49]. *XMemPod* implements remote memory paging using an Infiniband RDMA network, though it makes no assumption on specific RDMA network technology.

The Memory Paging Service Manager is the client module implemented on top of the *kswapd*, a default kernel daemon, responsible for memory paging. The PSM intercepts swap-out and swap-in operations and redirects them to *XMemPod*. The PSM provides a virtualized block device interface to the applications on a guest VM. This virtualized block device can be configured as a low latency primary swap device. Extensions to this virtualized block device is to configure it as a low latency volatile file system for storing large datasets, or a memory-mapped I/O space for large memory application. For the guest VM, the block device is simply a fast I/O partition (e.g., a linear I/O space) with an order of magnitude smaller access latency than disk. *XMemPod* implements the host-guest shared memory swap partition with both ramdisk based option and shared memory based option. The former is mounting a host resident ramdisk to the guest VM, and the latter uses a shared memory pool provided by the host, which is mapped into the host swap partition address space on guest VM. Comparing to the ramdisk solution, the shared memory approach delivers 1.7x better performance via a loadable kernel module for Linux 4.1.0 for all workloads we have tested. By default, *XMemPod* uses the shared memory solution.

The Shared Memory Manager (SMM) uses a POSIX shared memory region as the basis for sharing memory between host and VMs. On host, *XMemPod* is implemented and runs as a user space program, which manages host and remote shared memory block devices and disk block device, each device exposes a conventional block device I/O interface to the guest OS's VMA manager, which treats the block device as a fixed size swap partition. No modification is required to the host OS or any of its existing kernel modules, including KVM. The entire host shared memory area is divided into multiple elastic memory *pods*, one per VM on the host, provided that a shared memory pod is only established for a guest when paging initially occurs, and the shared memory pod is revoked from the guest when the execution of its applications terminates, or when the host shared memory manager terminates its service to the guest VM. Upon creating a shared memory pod, its entire address space is logically partitioned into slabs of fix size, according to `Host-SlabSize` in initial configuration. The shared memory pod manager maintains the mapping between the page offset in the guest swap partition and the address in its corresponding shared memory swap partition, and it opportunistically adjusts the size of each shared memory pod to decrement or increment one slab, based on the lower and upper pod utilization thresholds, defined by `sm-pod-min` and `sm-pod-max`. There are three types of pages in the host shared memory: *active* pages, those being used, *free* pages, those allocated to the pod but not yet being used, and *idle* pages, those that do not belong to any pod.

The Remote Memory Manager is designed with similar principles. First, the setting of `Remote-SlabSize`, `Remote-pod-min`, and `Remote-pod-max` can be different from those set for host shared memory configuration. Second, slabs are the units for balancing memory utilization across remote machines and providing low latency mapping to remote memory. All pages in a slab are mapped to the same remote memory. However, different slabs in the same remote pod can be mapped to multiple remote servers' memory. Third, we implement remote memory management on top of nbdX [1], which exploits the advantages of multi-queue implementation in the block layer and the Accelio acceleration facilities to provide fast IO to a remote shared memory device. When a paging request is redirected from a guest VM to the remote memory manager, it first selects one primary remote server and one secondary remote server from the `remote-count` candidates maintained by *XMemPod*, and uses the corresponding remote memory pods to expose the I/O interface of remote shared memory and use VirtIO to the guest (VM1) for read and write requests. For each remote server, a remote pod connector is setup to keep the RDMA connection with the remote server. On a remote node, when its NIC receives paging requests, it accesses the pages by mapping the target offset and size into the corresponding remote RDMA registered memory region. To guarantee the data to be delivered without corruption, Reliable Connected Queue Pair (RC QP) is configured on both sides. For swap-in request (read), the SMM passes to RMM the metadata from the compressed disaggregated memory page table (CSPT), including compressed page size, location of the compressed page in the remote memory pod(s), the instruction on proactive swap-in (batch read), RMM checks metadata parameters and performs swap-in operations. RMM also broadcasts periodic resource utilization messages which the client modules use to discover the available remote memory servers, such as their memory availability and load as well as page transfers to its clients. When a sever reaches its remote memory capacity `Remote-pod-max`, it will decline to serve any new swap-out (write) requests from its clients.

The Disk Swap Partition Manager is called to serve a swap-out request if there is insufficient remote memory in the cluster or if time-out occurs from an existing swap-out operation. DSM will resort to conventional OS swap facility to perform the swap. To ensure the availability of swap-out pages, *XMemPod* ensures the triple modular redundancy by maintaining three copies of each swap-out page in three different remote nodes. Although in the first implementation of the XMemPod, a swap-out event is committed only when at least three remote servers have committed the swap-out event. We do maintain triple modular redundancy for each swap page with the eventual consistency guarantee.

Connection Management. *XMemPod* uses one-sided RDMA READ/WRITE operations for data plane activities and RDMA SEND/RECEIVE operations for control plane activities. For each individual connection, two channels are established: (i) RDMA channel is used for maintaining the network connection, data transfer, and data corruption. (ii) *XMemPod* channel maintains status of remote MemPod by interacting with remote agent and uses it for placement and eviction algorithms.

Remote Memory Balancing. Several remote memory selection algorithms are implemented to minimize memory imbalance across machines and used as configuration

parameters, including round robin (RR), weighted RR, random, or power of two choices [24], [50], with RR as the default. *XMemPod* proactively allocates memory slabs of size `Remote-SlabSize` and registers them as memory regions for RDMA operations on remote servers, which reduces the cost of initialization process. Remote idle memory is monitored and when it drops below the `Remote-low` threshold, remote memory slabs will be deregistered preemptively through our remote slab eviction handler based on `Remote-SlabSize`, and updates the respective compressed swap page table (CSPT) corresponding to the deregistered slabs (see next section for detail on CSPT). At the same time, new remote memory servers will be selected to host the evicted pages for maintaining the dual replica of swap-out pages.

Fault Tolerance. *XMemPod* runs its client module of the disaggregated memory paging service on a guest VM, and its server modules, such as SMM, RMM, DSM on the host in every node of the cluster. If the corresponding guest VM or the host machine fails, or the local disk fails during paging, such as server/VM crash, *XMemPod* provides the same failure semantics as the guest OS swap facility today. For remote memory paging, it is important to handle unexpected failure scenarios due to network connection (link) failure or remote server failures. First, *XMemPod* does not require central coordination for remote memory paging, thus, single point of failure and frequent message synchronization are avoided. Second, each remote swap-out operation is replicated to at least two remote nodes and also maintained in the local disk swap of the guest VM. Third, each remote paging operation is treated as an atomic transaction, all or nothing, which is recorded in the corresponding entry of the CSPT, thus removing the inconsistency due to remote connection failure or remote server failure. Although in the first prototype, only dual remote servers are used, to support stronger fault tolerance, *XMemPod* may support paging to more than two remote servers and maintains the metadata in the corresponding CSPT for each swap-out request. This not only further increases the fault tolerance in the presence of network connection failure and unreachable server induced failure, but may also simplify the remote slab eviction handling.

Consistency. For each VM, we maintain a CSPT, which works as a log table to track of where a swap-out page is. We set the *CSPT_entry* using a location flag. When it is set to `LOCAL`, the page is stored in the host-VM shared memory pod, when it is set to `REMOTE`, the `REMOTE` tag will provide two location entries: the primary remote MemPod and the secondary remote MemPod. Otherwise, the local disk partition is used, indicating insufficient remote memory in the cluster or failure to reach remote servers before time-out. For swap-out request, *XMemPod* synchronously writes data into RDMA channel of primary remote MemPod and secondary remote MemPod. Once the RDMA WRITE operation of two remote MemPod completes, and its corresponding *CSPT_entry* is updated, the swap-out operation is committed. To avoid data corruption during transferring process, RC QP is configured for each RDMA channel, which guarantees that messages are delivered from a requester to a responder at most once and in order without correction. Requester considers a message operation complete once there is an ack from the responder that the message was read/written to its memory. Responder considers a message operation complete once the message was read/written to its memory. For swap-in request, if the *CSPT_entry* is set to `REMOTE`, *XMemPod* will put one RDMA READ operation on the primary MemPod address into the RDMA dispatch queue. When the READ completes, *XMemPod* responds by marking the commit of the read operation. Otherwise, *XMemPod* reads it from the secondary MemPod, or the local disk partition. We consider two failure cases.

1) *Local VM Failure/Exit*. When the PSM is unreachable, *XMemPod* considers the scenario as local VM failure or exit. Upon detecting this failure, the garbage collection is triggered at both local host-VM shared MemPod and two remote MemPods. It marks all slabs for this VM as unmapped and removes corresponding CSPT. The garbage collection for local disk is performed and cleared by guest OS.

2) *Host Failure*. For each server module of *XMemPod*, its RMM periodically checks the reachability of each remote MemPod agent in its remote server group, and considers unreachability of a remote agent as the remote host failure scenario. Upon detecting this failure, *XMemPod* updates the corresponding *CSPT_entry*(s) as unavailable. For a failed host, the remote agent unmaps and deregisters all slabs that are used as remote MemPod slabs for page requests read from and write to this failed host. The broken RDMA and *XMemPod* channel will be removed from the agent as well.

Scalability. A fundamental challenge for providing cluster-wide memory virtualization is to scale the system to terabytes of collective memory in a cluster. However, to track where each swap-out page is located in the cluster, the server module corresponding to the VM needs to maintain the metadata such as server ID, MemPod ID and offset in the CSPT entry of this page in either the host-VM shared memory or in the RDMA registered memory region for remote memory sharing. Consider the scenario that a simple in-memory hash table is used to store the location of each of its pages in the cluster, and each page is 4 KB in size. If we use 8 bytes to store each location identifier metadata, then we would need up to 5 GB host-VM shared memory to store the hash table for the 2 terabytes of cluster-wide memory. For 10 TB, it is 25 GB. Maintaining a CSPT of such size for each VM will incur prohibitively high cost as the cluster-wide memory scales up. To address this scalability challenge, *XMemPod* adopts group sharing model, where servers in a cluster are partitioned into groups of similar sizes. Servers within a group can share remote memory with one another. Each server can access the up-to-date memory sharing state of all other members of its group via its group leader. A leader election protocol [32] periodically elects the one with maximum available memory as the leader of a group. If the leader node crashes (handshake time-out), a new leader election process will be triggered. Also, leaders of all remote memory sharing groups form the top tier grouping service, which supports dynamic re-grouping upon request from their member client(s).

XMemPod is deployed successfully on small size RDMA clusters (32 machines) with up to 8 VMs each, a total of up to 256 VMs. As the number of VMs increases on a host, there is small overhead involved in dynamic allocation/

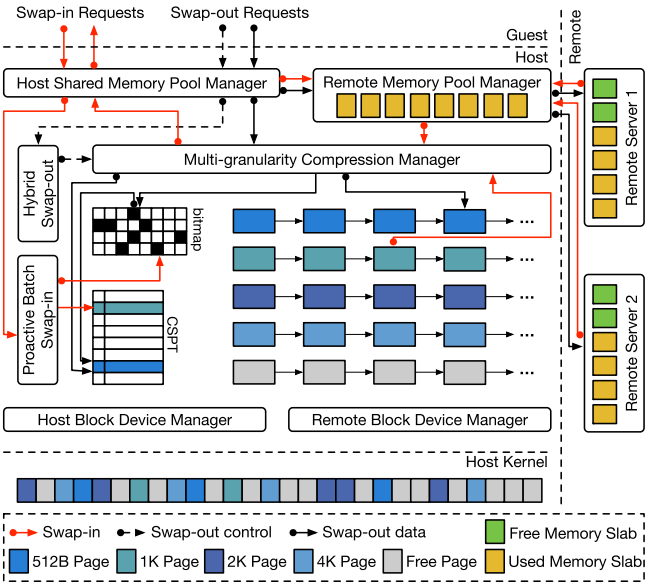


Fig. 4. *XMemPod* optimizations.

deallocation of shared memory pools in unit of slab. Right sizing of slabs is beneficial. Larger slabs can decrease the utilization efficiency of memory sharing. Also, the current sharing strategy for host and remote shared memory is greedy and demand driven.

4 *XMemPod* Optimizations

We introduce three additional optimizations to further improve the efficiency of *XMemPod*. Fig. 4 shows how they are orchestrated in concert for both local host and remote memory sharing.

4.1 Compressed Swap Page Table: CSPT

XMemPod compresses all swap-out pages before placing them into the proper swap partition. All swap-in pages are decompressed first before sent back to the application's working memory. Based on the notion that the compression efficacy mainly depends on two factors: compressibility of data and choice of data granularity [36], [52], *XMemPod* deploys a multi-granularity compression algorithm, which is optimized for achieving the best compressibility: e.g., a 4 KB page can be best compressed to one of the four different sizes: (0,512 B], (512,1 KB], (1 KB,2 KB], and (2 KB,4 KB]. LZ4 [19] is used as the default. Other multi-granularity compression algorithms include LZO-1X [43].

To keep track of compressed swap pages, four compressed page placement queues are created: for 512 B queue, 8 compressed page slots are created for one shared memory page, each holds a compressed page of size in (0,512 B]. Similarly, 4 compressed page slots for 1 KB queue, each slot holds a compressed page of size in (512 B, 1 KB], 2 slots for 2 KB queue in (1 KB, 2 KB], and 1 slot for 4 KB queue in (2 KB, 4 KB]. *XMemPod* creates and maintains a compressed disaggregated memory page table (CSPT) to keep track of each swap-out page and its metadata. With CSPT, the swap-in operation can easily lookup and locate the compressed page from the respective swap partition, and send it back to the guest in decompressed format. A hash-table based index structure [34], [44] is used to optimize read/write operations on the CSPT. When a swap-out operation is intercepted, a *CSPT_entry* is created for the swapped out page and links it to the Hashtable and a bitmap variable field is used to indicate the location of the swap partition where the compressed page currently resides. Other metadata recorded in the *CSPT_entry* includes the address of this compressed page, the length of compressed page which is required for decompression.

4.2 Hybrid Swap-Out

Upon receiving a swap-out request, the PSM intercepts the page and calls the *swap_writepage()* operation. The operation begins by compressing the page into a temporary buffer in order to know which one of the four compressed page-groups this compressed page should belong to. Based on the compressed page size, *XMemPod* chooses the shared memory page in the most suitable compressed page group, locates the free slot with its offset, and places the compressed page in the slot. Next, the PSM checks with SMM to determine if there is sufficient room in the host-VM shared memory pod to serve this swap-out request. When there is insufficient host memory, instead of putting the *most recent pages to remote* (MRPR), we advocates the hybrid swap-out optimization by placing *least recent pages to remote* (LRPR), which maximizes the utilization of host-VM shared memory to serve the most recent paging traffic, and selects remote memory as the next preferred tier for hybrid swap-out. Similarly, when there is insufficient remote memory in the cluster, the hybrid swap-out procedure will split the older pages by their LRU stamps and move the older ones to disk swap partition and make room in the remote memory pod for relatively more recent pages. In both cases, pages remain compressed during hybrid swap-out process and updates are performed on the compressed disaggregated memory page table and the hashtable index for remote memory accordingly.

4.3 Proactive Swap-in

We observe that when the working set of application is big, the page table grows and the PTE lookup cost increases. For frequent swap-in events, this cost can be non-trivial. *XMemPod* accelerates swap-in operations by piggyback of some metadata: when a page is swapped out, *XMemPod* keeps the address of the PTE related to this page as a piece of metadata together with the swapped-out page, and stores the metadata in the corresponding entry of the CSPT for this swap-out page. For each swap-out page, this metadata only takes up 4 bytes. The cost of keeping this metadata in the swap area is only about 1/1000 of the total size of the swapped-out pages. Using this metadata, when a page is swapped into the working memory of the guest, *XMemPod* is able to quickly locate the PTE that needs to be updated by referring to this metadata without the need to scan the page table. Thus, the time spent on accessing the PTE of a swapped page will not increase as the size of the system wide page table grows. This advantage is applied to proactive swap-in from host to guest, from remote to guest, or from remote to host.

When swap-in a page from remote memory pods, the SSM will check if the free pages in the shared memory pod

TABLE 1
Applications Used in Experiments

Workload	Suite/Application	Dataset
PageRank	Spark GraphX	1 million pages
LogisticRegression	Spark Mllib	7.5 million samples
TunkRank	PowerGraph	30 million vertices
Kmean	PowerGraph	7 million samples
SVM	Liblinear	45 thousand samples
YCSB-Memcached	Memcached	20 million records
YCSB-Redis	Redis	20 million records
YCSB-VoltDB	VoltDB	20 million records
Canneal	PARSEC	2.5 million records
Apache Solr	Cloudsuite	12 GB index

TABLE 2
Machine Learning Workload Performance Comparison of Linux, nbdX and *XMemPod* – Completion Time (second)

		Page Rank	Logistic Regression	Tunk Rank	Kmean	SVM
100%	Linux	226	520	319	216	536
	nbdX	227	519	318	213	540
	XMemPod	228	512	319	215	538
75%	Linux	19800	12240	2830	847	1512
	nbdX	302	869	798	451	874
	XMemPod	**238 (83x)**	**545(22x)**	**352(8x)**	**246(3x)**	**585 (3x)**
50%	Linux	25200	18360	4273	22320	15671
	nbdX	1263	1298	954	712	1152
	XMemPod	**297(85x)**	**694(26x)**	**403(11x)**	**284 (79x)**	**691 (23x)**

are above the pre-defined batch swap-in threshold, and if yes, the remote memory pod manager will trigger proactive batch swap-in, which swaps in the requested page together with those nearby based on temporal or spatial locality. The batch swap-in threshold is a XMemPod configuration parameter that is initialized at the system startup time and can be modified by the shared memory manager based on the available free memory at runtime. This batch threshold parameter is defined by a pair of values, the first value specifies the number of pages to be included in the proactive batch when a page is being swapped in, and the second value specifies which pages are chosen with respect to the swap-in page in terms of the temporal or spatial locality. In the first prototype implementation of XMemPod, we only implement spatial locality. When the #page threshold is defined to be k, only those k pages that are near by the page being swapped-in will be included in the batch.

5 Evaluation

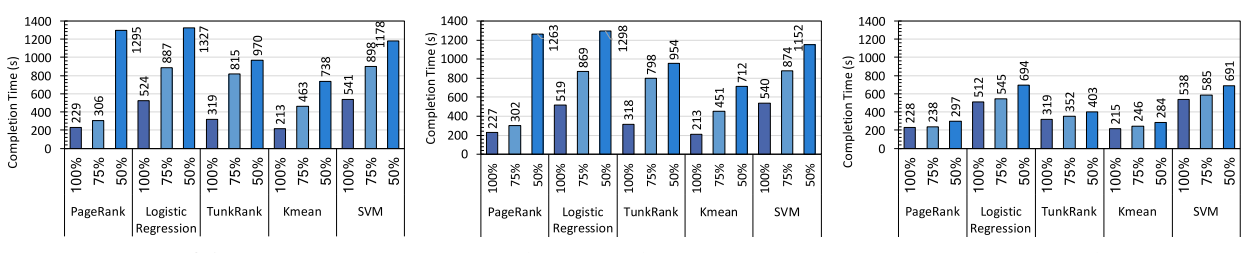
We evaluate *XMemPod* using ten popular memory-intensive applications, with first five machine learning and next five big data applications, listed in Table 1, and compare *XMemPod* with Linux (conventional OS swap facility), Accelio nbdX [1] and Infiniswap [24].

Experiments are performed on a 32-machine, 56 Gbps Infiniband cluster. Each machine has 32 core E5-2650v2 CPU, 64 GB memory, 2 TB SATA 7.2K rpm hard drives, and running KVM 1.2.0 with QEMU 2.0.0 as virtualization platform. We use Linux 4.1.0 and Ubuntu 14.04 for both the guest and host system. For most of the experiments unless otherwise stated, we run 80 VMs on a 32-machine RDMA cluster and created an equal number of VMs for each application workload. We started with the 100 percent configuration by creating VMs with large enough memory to fit entire workload in memory. We measured the peak memory consumption, and then ran 75 and 50 percent configurations by creating VMs with enough memory to fit these fractions of the peak memory consumption. The working sets for the ten applications range from 25 GB to 30 GB and their input dataset sizes range from 12 GB to 20 GB per VM. Unless explicitly stated, we use the default configurations: `sm-SlabSize=512 MB`, `Remote-SlabSize=1 GB`, hybrid swap-out threshold 80 percent, proactive swap-in threshold 50 percent, and remote server connection timeout is `1s`.

For fair comparison, the setups for nbdX, Infiniswap and *XMemPod* use the RDMA registered memory region of the same size for remote memory sharing on each node. In comparison, the Linux setup does not reserve RDMA buffer for remote memory and utilizes the physical memory on each node for its hosted VMs. For the VMs on a local node, *XMemPod* consolidates the total amount of allocated memory per VM to create the host-coordinated shared memory pods at the cost of assigning smaller memory to each of its VMs.

5.1 Impact on Applications

Machine Learning Applications Performance. We first measure and compare the performance of the three systems on PageRank, LogisticRegression, TunkRank, Kmean, and SVM as shown in Table 2. We also compare Infiniswap, *XMemPod Remote*, and *XMemPod*, as shown in Figs. 5a, 5b, and 5c. We highlight three observations: First, for 75 percent configuration, *XMemPod* improves application completion time by 24x on average and up to 83x over Linux, and improves over Infiniswap and nbdX by 2.3x and 2.2x respectively. Second, for 50 percent configuration, *XMemPod* improves application completion time by 45x on average and up to



(a) Infiniswap (b) XMemPod Remote (c) XMemPod

Fig. 5. Machine learning workload performance comparison of infiniswap, *XMemPod Remote* and *XMemPod*.

TABLE 3
Big Data Workload Performance Comparison of Linux, nbdX and *XMemPod* – Completion Time (second)

		Memcached		Redis		VoltDB		Canneal	Solr
		ETC	SYS	ETC	SYS	ETC	SYS		
100%	Linux	397	424	250	310	858	928	113	60
	nbdX	398	423	251	311	858	925	113	60
	XMemPod	397	424	250	310	857	926	112	60
75%	Linux	23640	23781	112359	128205	44943	62305	3623	2216
	nbdX	1630	1669	1475	1585	1545	1619	251	839
	XMemPod	**451(52x)**	**466(51x)**	**265(424x)**	**321(399x)**	**904(50x)**	**993(63x)**	**117(31x)**	**65(34x)**
50%	Linux	84745	85470	158730	217391	235394	333333	6806	3649
	nbdX	2415	2493	2918	3346	1790	1807	479	961
	XMemPod	**494(172x)**	**509(168x)**	**277(573x)**	**355(612x)**	**979(240x)**	**1057(315x)**	**167(41x)**	**72(51x)**

85x over Linux, and improves over Infiniswap by 4.4x (best case) and 2.6x on average, improves over nbdX by 4.3x (best case) and 2.5x on average. Third, For 75 percent configuration, using *XMemPod*, machine learning applications experience only on average 1.1x increase in completion time, comparing to 1.8x using nbdX, 1.9x using Infiniswap and 25x using Linux. For 50 percent configuration, using *XMemPod*, machine learning applications experience only on average 1.3x increase in completion time instead of 2.5x using nbdX, 3.4x using Infiniswap and 59x using Linux.

Big Data Applications Performance. We next measure and compare the three systems on five big data applications: Memcached [7], Redis [8], VoltDB [9], Canneal [3], and Apache solr [2]. For Memcached, Redis and VoltDB, we use the dataset published by Facebook [14] and choose ETC and SYS as workloads to explore different rates of SET operations. ETC is read-intensive workload, which has 5 percent SET and 95 percent GET, and SYS has 25 percent SET and 75 percent GET, which is popularly used as write-intensive workload [14], [24]. Table 3 shows the completion time for vanilla Linux, nbdX, and *XMemPod*. Figs. 6a, 6b, and 6c show the comparison of Infiniswap, *XMemPod Remote*, and *XMemPod*.

We observe three interesting results: (1) For 75 percent configuration, *XMemPod* improves application completion time over Linux by 138x on average and 423x in the best case, and improves over Infiniswap and nbdX by 13.2x and 12.9x in the best case, and 4.7x and 4.5x on average, respectively. (2) For 50 percent configuration, *XMemPod* improves application completion time over Linux by 612x in the best case and 271x on average. It improves application completion time by 13.8x and 13.3x in the best case and 6.6x and 6.2x on average over Infiniswap and nbdX respectively. (3) Applications using *XMemPod* benefit from more steady performance, but experience high degree of fluctuations in performance when using Infiniswap or nbdX.

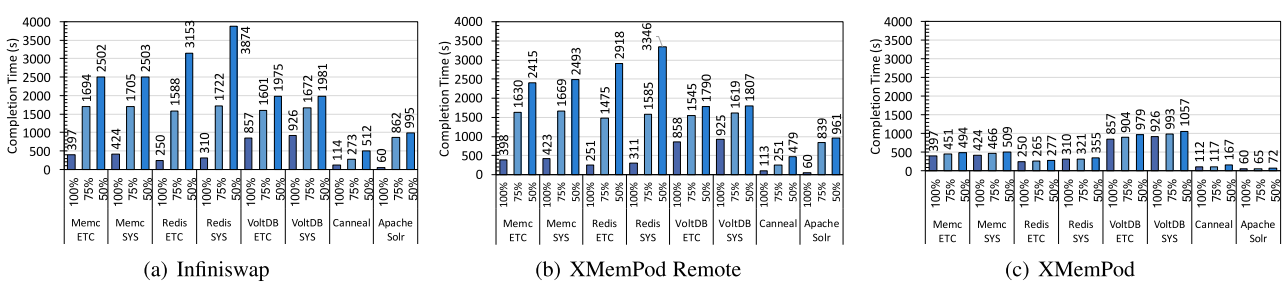
For 50 percent configuration, using *XMemPod*, big data applications experience on average 1.2x increase in completion time, compared to 7.9x using Infiniswap, 7.3x using nbdX and 313x using Linux.

Given that tail-latency is considered a more accurate indicator of performance degradation for bigdata workloads, in addition to median latency, we also measured the 99th percentile response latency for Memcached, Redis and VoltDB respectively for all four systems, as shown in Fig. 7. Using *XMemPod*, the 99th percentile latencies for all applications are much closer to the performance of 100 percent configuration when 50 percent of their working sets do not fit in memory. For example, for 50 percent configuration, using *XMemPod*, the 99th tail latencies of Memcached, Redis, and VoltDB are increased by 42, 16, and 33 percent respectively. In contrast, the 99th tail latencies are increased by 19.7x, 16.0x, and 3.4x respectively by using Infiniswap, and by 16.9x, 13.9x, and 2.5x respectively by using nbdX.

5.2 Cluster Utilization

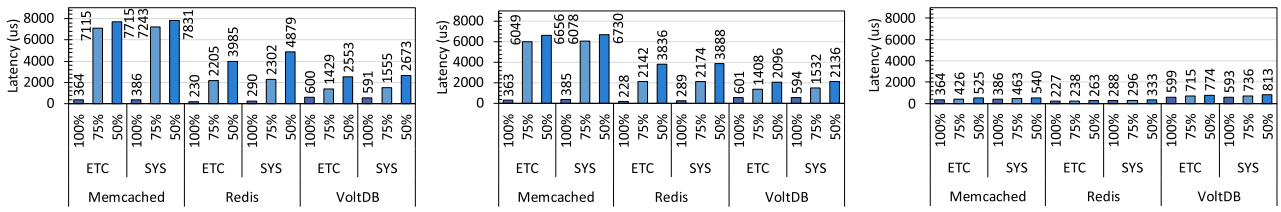
Fig. 8 shows the amount of send/receive data over the RDMA cluster of 32 machines. The total amount of network traffic over RDMA in the case of Infiniswap is 524 GB, while *XMemPod* is 128 GB, which is only 24 percent of Infiniswap. Similar comparison is observed for nbdX.

Fig. 9 compares the overall data distribution with respect to data stored in local host versus data stored in remote nodes for *XMemPod* and Infiniswap. It is interesting to observe that for Infiniswap, the ratio of the average of data stored in local versus remote is 65 percent versus 35 percent.



(a) Infiniswap (b) XMemPod Remote (c) XMemPod

Fig. 6. Big data workload performance comparison of infiniswap, *XMemPod Remote* and *XMemPod*.



(a) Infiniswap (b) XMemPod Remote (c) XMemPod

Fig. 7. Big data workload 99th percentile latency comparison of infiniswap, *XMemPod Remote* and *XMemPod*.

In contrast, for *XMemPod*, the ratio of average of data stored in local versus remote is 90 percent versus 10 percent.

5.3 Host/Remote Memory Distribution

We evaluate the impact of hierarchical memory orchestration of *XMemPod* on Redis, Memcached, VoltVB by varying its host and remote memory distribution using 50 percent configuration. XM-SM denotes 100 percent of paging events are handled in host shared memory. XM-RDMA denotes 100 percent in remote memory via RDMA network, and zero host shared memory is reserved by *XMemPod*. The remaining three host-remote distribution split ratios are XM-9:1, XM-7:3, and XM-5:5. XM-9:1 denotes that 90 percent of paging traffic is served in host-VM shared memory pod and 10 percent of it is sent to the remote memory pods. We compare the performance of *XMemPod* using varying distribution configurations with Linux, Infiniswap and nbdX.

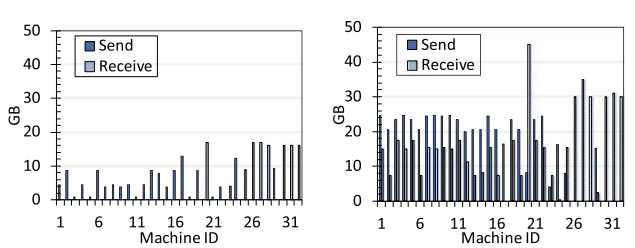
Fig. 10 shows the results. We highlight four observations. *First*, Fig. 10a shows that using XM-SM, throughputs of Redis, Memcached, and VoltDB increase by up to 571x, 171x, and 240x respectively, compared with Linux, increase by 11.4x, 5.1x, and 2.0x compared with Infiniswap, and increase by 10.5x, 4.9x, and 1.8x compared with nbdX. *Second*, using XM-RDMA, throughput of Redis, Memcached and VoltDB increase by 3.2x, 1.8x, and 1.6x respectively, compared to Infiniswap, and increase by 2.9x, 1.8x, and 1.5x respectively, compared to nbdX. *Third*, Fig. 10a shows that the throughputs of all three applications reduce, as the percentage of remote memory increases for *XMemPod* from XM-SM, XM-9:1, XM-7:3, XM-5:5 to XM-RDMA. Figs. 10b and 10c show that the median and 99th percentile latencies using *XMemPod* are significantly shorter for Redis, Memcached, and VoltDB compared to Linux, Infiniswap and nbdX for all distribution settings of host-remote memory sharing. Even with remote only XM-RDMA, the 99th percentile latencies of Redis, Memcached and VoltDB improve by 1.8x, 1.7x, and 3.1x, comparing to Infiniswap, and by 1.7x, 1.5x, and 2.6x, comparing to nbdX.

5.4 Impact of Remote Memory Sharing

In this section, we deploy *XMemPod* on a 32-machine RDMA cluster on Emulab [5]. We turn off the host-VM shared memory option and only rely on remote memory for serving paging requests under 50 percent configuration. We randomly choose a server with two VMs running Memcached and Redis ETC workload respectively, and configure their remote memory slabs to be distributed across multiple remote servers. We measure the performance of remote paging to 2 to 8 servers, each of which runs a VM for Logistic Regression workload. We report two sets of experiments due to space constraint.

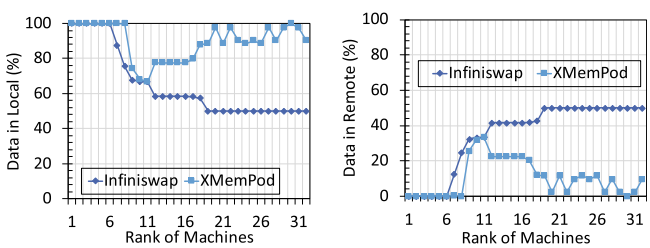
Remote Paging to Multiple Servers. Fig. 11 measures the impact of remote paging on both host and remote application performance. We make three observations. *First*, on host server, using *XMemPod*, all applications complete significantly faster than the scenario without *XMemPod*. Furthermore, *XMemPod* with 2 remote server case improves the completion time of Memcached and Redis by 70x and 179x respectively, over the scenario without *XMemPod*. *Second*, the performance of both Redis and Memcached is stable as we increase the number of servers for serving remote paging. This is because each server only equips with one Infiniband card in the Emulab RDMA cluster, the performance of remote read/write operation is limited by one network card. We expect that *XMemPod* will benefit from parallel remote I/O if multiple Infiniband cards are equipped. *Third*, Logistic Regression on each of the remote servers serving remote paging traffic from Redis and Memcached exhibits stable performance in all cases, which is expected because *XMemPod* uses one-sided RDMA operation to read/write remotely, which completely bypasses remote CPUs.

Impact of Eviction on Application Performance. Fig. 12 measures the impact of remote memory eviction operations on the performance of both host and remote applications. We highlight two observations. First, eviction increases the completion time of Memcached and Redis by 1.3 and 1.4 percent. A primary factor is caused by the process of



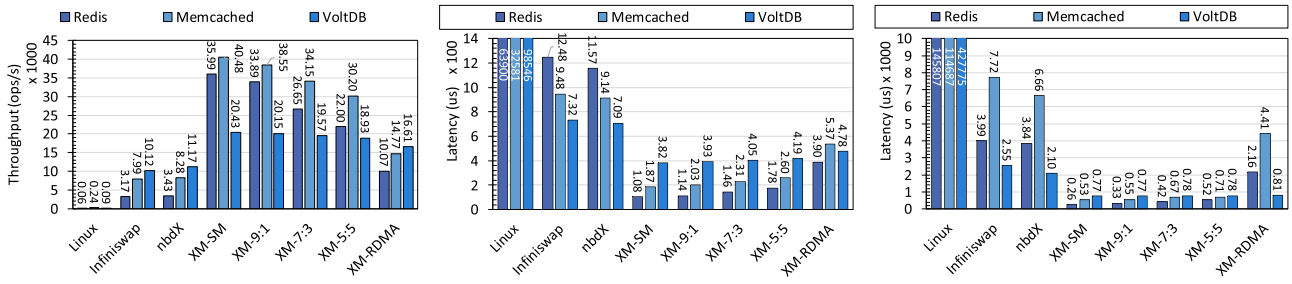
(a) XMemPod Send/Receive Data (b) Infiniswap Send/Receive Data

Fig. 8. Network traffic of individual machines.



(a) Data Stored in Local (%) (b) Data Stored in Remote (%)

Fig. 9. Data distribution of individual machines.



(a) Throughput (b) Median Latency (c) 99th Latency

Fig. 10. Varying host and remote memory sharing distribution (different latency scale in Fig. 10b and 10c).

EVICT event handling, updating CSPT (Section 4.1), and redirecting request to the other remote server that stores the copy of requested memory slab. Second, eviction has no performance impact on Logistic Regression on remote server, as shown in Fig. 12b. The main cost of evicting slab is memory region deregistration. For 1 GB slab, the average cost of it is 273 microseconds. When we increase to 4 GB, 8 GB or 16 GB of slabs, the average cost for memory region deregistration is 1,121, 2,157, and 4,356 microseconds.

6 Related Work

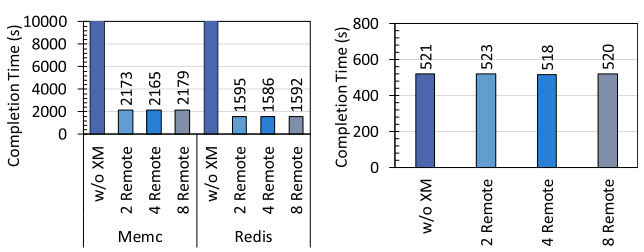
Dynamic Memory Balancing. Existing research on scheduling and consolidating physical memory among VMs centered on dynamic memory balancing, with Ballooning [56] as the most representative. Proposals [26], [62] devoted to periodic estimation of VM working set size. But, accurate working set prediction is hard under changing workloads [30].

Memory Page Compression and Deduplication. Existing proposals for page deduplication [27] aim at identifying duplicate memory pages. Some open source efforts provide transcendent memory (tmem) interface for paging [6] or a compressed RAM cache [33]. Proposals for addressing double paging [11] and memory compression [46], [51] can be leveraged in *XMemPod* to further improve host/remote memory sharing.

Distributed Shared Memory/Disaggregated Memory. Distributed shared memory was studied extensively [12], [15]. However, DSM suffers poor performance due to high communication overhead. Disaggregated memory has attracted much attention recently [13]. Disaggregated remote memory has attracted much attention over the past decade [18], [22], [24], [28], [38], [39], [40]. Most proposals rely on new hardware architecture, new network protocols to cut down the communication cost. Recently, some proposals show the benefit of leveraging RDMA technology [25], [41], such as remote storage for key-value stores[21], distributed objects [55], swap pages [24], object replication [61], and disaggregated datacenters in a box [4]. Most of these efforts lack of desired transparency. Infiniswap [24] is developed on top of Accelio nbdX [1], and both use RDMA read/write interface for fast and reliable access to remote memory regions registered for RDMA networking. But, existing solutions via RDMA access to remote memory cannot be used to implement host-coordinated shared memory sharing among VMs on the same host. To the best of our knowledge, *XMemPod* is the first to implement local memory sharing between VMs on the same host and to promote a hierarchical memory sharing framework with host-coordinated memory sharing first, followed by RDMA-coordinated remote memory sharing. Our experiments show that the *XMemPod* approach to hierarchical orchestration of disaggregated memory is highly effective.

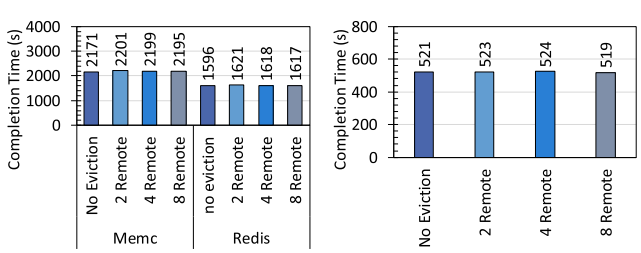
7 Conclusion

We have presented *XMemPod*, a hierarchical disaggregated memory orchestration framework for virtualization of cluster-wide memory. *XMemPod* is the first to implement memory-speed access to available memory in other VMs on the same host, and to promote a hierarchical memory sharing framework with host-coordinated memory sharing first, followed by RDMA-coordinated remote memory sharing. *XMemPod* is deployed on a virtualized RDMA cluster without any modifications to user applications and the host OSes. It works as a loadable kernel module to dynamically extend running OS to support *XMemPod* functionality. It can also be unloaded to free the memory and other resources when the functionality is no longer required. It offers orders of magnitude performance gain for bigdata and ML applications. *XMemPod* incurs no functional nor performance impact on guest VMs that do not use it or that incurs zero paging. Evaluated on unmodified Spark, Apache Hadoop, Memcached, Redis and VoltDB, *XMemPod* improves throughputs of these



(a) Host - Memcached and Redis (b) Remote - Logistic Regression

Fig. 11. *XMemPod* with multiple remote servers.



(a) Host - Memcached and Redis (b) Remote - Logistic Regression

Fig. 12. Eviction impact on performance.

applications over Linux by 11x to 612x, improves median and tail latencies by 174x to 702x and 218x to 630x respectively. Compared to the state of art remote memory paging systems (Infiniswap and nbdX), *XMemPod* improves throughputs by up to 14x, median latencies and tail latencies by up to 12x and 15x respectively. *XMemPod* is released at https://github.com/git-disl/XMemPod.

Acknowledgments

This research has been partially supported by the National Science Foundation under Grants IIS-0905493, CNS-1115375, NSF 1547102, SaTC 1564097, and Intel ISTC on Cloud Computing. Any opinions, findings, and conclusions or recommendations expressed in this article are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Accelio based network block device, 2015. [Online]. Available: https://github.com/accelio/NBDX
[2] Apache Solr, 2005. [Online]. Available: http://lucene.apache.org/solr
[3] Canneal, 2008. [Online]. Available: http://parsec.cs.princeton.edu/overview.htm
[4] dReDBox: Disaggregated data center in a box, 2018. [Online]. Available: http://www.dredbox.eu/publications/
[5] Emulab, 2008. [Online]. Available: https://www.emulab.net
[6] Frontswap, 2012. [Online]. Available: https://www.kernel.org/doc/Documentation/vm/frontswap.txt
[7] Memcached, a distributed memory object caching system, 2003. [Online]. Available: https://memcached.org
[8] Redis, an in-memory data structure store, 2009. [Online]. Available: https://redis.io
[9] VoltDB, a translytical in-memory database, 2010. [Online]. Available: https://github.com/VoltDB/voltdb
[10] M. K. Aguilera *et al.*, "Remote memory in the age of fast networks," in *Proc. Symp. Cloud Comput.*, 2017, pp. 121–127.
[11] N. Amit, D. Tsafrir, and A. Schuster, "VSwapper: A memory swapper for virtualized environments," *ACM SIGPLAN Notices*, vol. 49, pp. 349–366, 2014.
[12] C. Amza *et al.*, "TreadMarks: Shared memory computing on networks of workstations," *J. Comput.*, vol. 29, no. 2, pp. 18–28, Feb. 1996.
[13] K. Asanovic and D. Patterson, "FireBox: A hardware building block for 2020 warehouse-scale computers," in *Proc. USENIX Conf. File Storage Technol.*, 2014, pp. 1–46.
[14] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 40, pp. 53–64, 2012.
[15] J. K. Bennett, J. B. Carter, and W. Zwaenepoel, "Munin: Distributed shared memory based on type-specific memory coherence," in *Proc. 2nd ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 1990, pp. 168–176.
[16] R. Birke, L. Y. Chen, and E. Smirni, "Data centers in the wild: A large performance study," IBM Research, Zurich, Switzerland, 2012.
[17] P. Bodík, I. Menache, M. Chowdhury, P. Mani, D. A. Maltz, and I. Stoica, "Surviving failures in bandwidth-constrained datacenters," in *Proc. ACM SIGCOMM Conf. Appl. Technol. Archit. Protocols Comput. Commun.*, 2012, pp. 431–442.
[18] W. Cao and L. Liu, "Dynamic and transparent memory sharing for accelerating big data analytics workloads in virtualized cloud," in *Proc. IEEE Int. Conf. Big Data*, 2018, pp. 191–200.
[19] Y. Collet, "LZ4–Extremely fast compression," Tech. Rep., 2015.
[20] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 143–154.
[21] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro, "FaRM: Fast remote memory," in *Proc. 11th USENIX Conf. Netw. Syst. Des. Implementation*, 2014, pp. 401–414.
[22] P. X. Gao *et al.*, "Network requirements for resource disaggregation," in *Proc. 12th USENIX Conf. Operating Syst. Des. Implementation*, 2016, pp. 249–264.
[23] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proc. 8th USENIX Conf. Netw. Syst. Des. Implementation*, 2011, pp. 323–336.
[24] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with INFINISWAP," in *Proc. 14th USENIX Conf. Netw. Syst. Des. Implementation*, 2017, pp. 649–667.
[25] C. Guo *et al.*, "RDMA over commodity ethernet at scale," in *Proc. ACM SIGCOMM Conf.*, 2016, pp. 202–215.
[26] F. Guo, "Understanding memory resource management in vmware vsphere 5.0," VMware, Inc, Palo Alto, CA, 2011.
[27] D. Gupta *et al.*, "Difference engine: Harnessing memory redundancy in virtual machines," *Commun. ACM*, vol. 53, pp. 85–93, 2010.
[28] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker, "Network support for resource disaggregation in next-generation datacenters," in *Proc. 12th ACM Workshop Hot Top. Netw.*, 2013, pp. 1–7.
[29] M. Hao, G. Soundararajan, D. R. Kenchammana-Hosekote, A. A. Chien, and H. S. Gunawi, "The tail at store: A revelation from millions of hours of disk and SSD deployments," in *Proc. 14th USENIX Conf. File Storage Technol.*, 2016, pp. 263–276.
[30] M. R. Hines, A. Gordon, M. Silva, D. Da Silva, K. Ryu, and M. Ben-Yehuda, "Applications know best: Performance-driven memory overcommit with Ginkgo," in *Proc. IEEE 3rd Int. Conf. Cloud Comput. Technol. Sci.*, 2011, pp. 130–137.
[31] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis," in *Proc. IEEE 26th Int. Conf. Data Eng. Workshops*, 2010, pp. 41–51.
[32] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for internet-scale systems," in *Proc. USENIX Conf. USENIX Annu. Techn. Conf.*, 2010, Art. no. 11.
[33] S. Jennings, "The zswap compressed swap cache," in *LWN.net*, 2013. [Online]. Available: https://lwn.net/Articles/537422/
[34] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen, "Raising the bar for using GPUs in software packet processing," in *Proc. 12th USENIX Conf. Netw. Syst. Des. Implementation*, 2015, pp. 409–423.
[35] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kVM: The Linux virtual machine monitor," in *Proc. Linux Symp.*, 2007, pp. 1–6.
[36] M. Kjelso, M. Gooch, and S. Jones, "Empirical study of memory-data: Characteristics and compressibility," *IEE Proc.-Comput. Digital Techn.*, vol. 145, no. 1, pp. 63–67, Jan. 1998.
[37] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura, "SparkBench: A comprehensive benchmarking suite for in memory data analytic platform spark," in *Proc. 12th ACM Int. Conf. Comput. Frontiers*, 2015, pp. 1–8.
[38] S. Liang, R. Noronha, and D. K. Panda, "Swapping to remote memory over InfiniBand: An approach using a high performance network block device," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2005, pp. 1–10.
[39] 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 *Proc. Annu. Int. Symp. Comput. Architecture*, 2009, pp. 267–278.
[40] K. Lim *et al.*, "System-level implications of disaggregated memory," in *Proc. IEEE Int. Symp. High-Perform. Comp Archit.*, 2012, pp. 1–12.
[41] F. Mietke, R. Baumgartl, R. Rex, T. Mehlan, T. Hoefler, and W. Rehm, "Analysis of the memory registration process in the mellanox InfiniBand software stack. 8 2006," in *Proc. Eur. Conf. Parallel Process.*, 2006, pp. 124–133.
[42] J. Nelson *et al.*, "Latency-tolerant software distributed shared memory," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, 2015, pp. 291–305.
[43] M. Oberhumer, "LZO real-time data compression library," User Manual for LZO, 2008. [Online]. Available: http://www.oberhumer. com/opensource/lzo/
[44] R. Pagh and F. F. Rodler, "Cuckoo hashing," *J. Algorithms*, vol. 51, pp. 122–144, 2004.
[45] O. Paz, "InfiniBand essentials every HPC expert must know," in *Retrieved August*, 2014. [Online]. Available: http://www.hpcadvisorycouncil.com/events/2014/swiss-workshop/presos/Day_1/1_Mellanox.pdf
[46] G. Pekhimenko *et al.*, "Linearly compressed pages: A low-complexity, low-latency main memory compression framework," in *Proc. 46th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2013, pp. 172–184.

[47] R. Power and J. Li, "Piccolo: Building fast, distributed programs with partitioned tables," in *Proc. 9th USENIX Conf. Operating Syst. Des. Implementation*, 2010, pp. 293–306.
[48] 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 *Proc. 3rd ACM Symp. Cloud Comput.*, 2012, pp. 1–13.
[49] C. A. Reiss, "Understanding memory configurations for in-memory analytics," PhD thesis, Dept. Elect. Eng. Comput. Sci., Univ. California, Berkeley, Berkeley, CA, 2016.
[50] A. W. Richa, M. Mitzenmacher, and R. Sitaraman, "The power of two random choices: A survey of techniques and results," *Combinatorial Optim.*, pp. 255–304, 2001.
[51] S. Roy, R. Kumar, and M. Prvulovic, "Improving system performance with compressed memory," in *Proc. 15th Int. Parallel Distrib. Process. Symp.*, 2001, pp. 7 pp.-.
[52] R. B. Tremaine, T. B. Smith, M. Wazlowski, D. Har, K.-K. Mak, and S. Arramreddy, "Pinnacle: IBM MXT in a memory controller chip," *IEEE Micro*, vol. 21, no. 2, pp. 56–68, Mar./Apr. 2001.
[53] V. K. Vavilapalli *et al.*, "Apache hadoop YARN: Yet another resource negotiator," in *Proc. 4th Annu. Symp. Cloud Comput.*, 2013, pp. 1–16.
[54] J. Vienne, J. Chen, M. Wasi-Ur-Rahman, N. S. Islam, H. Subramoni, and D. K. Panda, "Performance analysis and evaluation of InfiniBand FDR and 40GigE RoCE on HPC and cloud computing systems," in *Proc. IEEE 20th Annu. Symp. High-Perform. Interconnects*, 2012, pp. 48–55.
[55] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall, "A note on distributed computing," in *Proc. Int. Workshop Mobile Object Syst.*, 1996, pp. 49–64.
[56] C. A. Waldspurger, "Memory resource management in VMware ESX server," *ACM SIGOPS Operating Syst. Rev.*, vol. 36, pp. 181–194, 2002.
[57] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, and M. D. Corner, "Memory buddies: Exploiting page sharing for smart colocation in virtualized data centers," *ACM SIGOPS Operating Syst. Rev.*, vol. 43, pp. 31–40, 2009.
[58] M. Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Conf. Netw. Syst. Des. Implementation*, 2012, Art. no. 2.
[59] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. 2nd USENIX Conf. Hot Top. Cloud Comput.*, 2010, Art. no. 10.
[60] P. Zhang, X. Li, R. Chu, and H. Wang, "HybridSwap: A scalable and synthetic framework for guest swapping on virtualization platform," in *Proc. IEEE Conf. Comput. Commun.*, 2015, pp. 864–872.
[61] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson, "Mojim: A reliable and highly-available non-volatile memory system," *ACM SIGPLAN Notices*, vol. 50, pp. 3–18, 2015.
[62] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar, "Dynamic tracking of page miss ratio curve for memory management," *ACM SIGOPS Operating Syst. Rev.*, vol. 39, pp. 177–188, 2004.

Wenqi Cao received the master's degree from the Computer Science and Engineering Department, Pennsylvania State University, State College, Pennsylvania, and has five years work experience as software engineer working for SIEMENS, HP, and Microsoft. He is currently working toward the PhD degree in computer science and is a research assistant with the School of Computer Science, Georgia Institute of Technology, Atlanta, Georgia. His research interest includes cloud computing, distributed system, and operating system.

Ling Liu (Fellow, IEEE) is a professor with the School of Computer Science, Georgia Institute of Technology, Atlanta, Georgia. She directs the research programs in Distributed Data Intensive Systems Lab (DiSL), examining various aspects of large scale data intensive systems, including performance, availability, security and privacy. She is an internationally recognized expert in the areas of big data systems and analytics, database systems, distributed computing, internet data management, and service oriented computing. She has published more than 300 international journal and conference articles and is a recipient of the best paper award from a number of top venues.