Journal of Cloud Computing:
Advances, Systems and Applications

RESEARCH

Open Access

Memory sharing for handling memory overload on physical machines in cloud data centers

Yaozhong Ge[1], Yu-Chu Tian[1*], Zu-Guo Yu[2] and Weizhe Zhang[3,4]

Abstract

Over-committing computing resources is a widely adopted strategy for increased cluster utilization in Infrastructure as a Service (IaaS) cloud data centers. A potential consequence of over-committing computing resources is memory overload of physical machines (PMs). Memory overload occurs if memory usage exceeds a defined alarm threshold, exposing running computation tasks at a risk of being terminated by the operating system. A prevailing measure to handle memory overload of a PM is live migration of virtual machines (VMs). However, this not only consumes network bandwidth, CPU, and other resources, but also compels a temporary unavailability of the VMs being migrated. To handle memory overload, we present a memory sharing system in this paper for PMs in cloud data centers. With memory sharing, a PM automatically borrows memory from a remote PM when necessary, and releases the borrowed memory when memory overload disappears. This is implemented through swapping inactive memory pages to remote memory resource. Experimental studies conducted on InfiniBand-networked PMs show that the memory sharing system is fully functional. The measured throughput and latency are around 929 Mbps and 1.3 μs, respectively, on average for remote memory access. They are similar to those from accessing a local-volatile memory express solid-state drive, and thus are promising in real applications.

Keywords Cloud computing services, Data center, Memory overload, Memory sharing, Resource over-committing

Introduction

Over-committing computing resources is a widely adopted strategy in Infrastructure as a Service (IaaS) cloud data centers for increased cluster utilization. In IaaS cloud data centers, virtual machines (VMs) usually do not always fully utilize their provisioned resources. The existence of a gap between the provisioned and actual utilized resources gives cloud service providers an opportunity to over-commit resources while meeting their service-level agreements (SLAs). Appropriate resource over-commitment enables the use of fewer physical machines (PMs). This will significantly reduce the operating costs for the data center, while only undertaking a low risk of violating Quality of Service (QoS) requirements. While over-committing resource has impacts on the performance of VMs, it shows limited performance degradation in general to the end users of cloud services except for memory overload.

Memory overload occurs when a PM has memory pressure over a threshold for more than five minutes [3]. It places running computation tasks under a risk of being terminated by the operating system. If the memory usage

*Correspondence:
Yu-Chu Tian
y.tian@qut.edu.au
[1] School of Computer Science, Queensland University of Technology, Brisbane, Australia
[2] Key Laboratory of Intelligent Computing and Information Processing of Ministry of Education, and Hunan Key Laboratory for Computation and Simulation in Science and Engineering, Xiangtan University, Xiangtan, China
[3] School of Cyberspace Science, Harbin Institute of Technology, Harbin, China
[4] Cyberspace Security Research Center, Peng Cheng Laboratory, Shenzhen, China

of a PM reaches its total capacity, the latest process that requests non-existent memory resources will be terminated. In this case, the SLA and QoS requirements may be violated. Therefore, over-committing the memory resources of PMs in a cloud data center demands a mechanism to deal with the memory overload problem.

A prevailing measure to handle memory overload of a PM is live VM migration. If a PM is considered as being overloaded, one or more VMs running on the overloaded PM could be migrated to other available PMs in order to avoid any violation of QoS requirements. However, live VM migration introduces overheads to all involved entities and processes, such as the source and destination PMs, the VM being migrated, and the VMs co-located on these two PMs [45]. The overheads can be a short unavailability to the VM being migrated, extra costs of shared infrastructure resources (e.g., network bandwidth), and/or extra costs of CPU resources on the source and destination PMs. In addition, the CPU of the source PM could be underutilized after the VM migration, reducing the CPU utilization.

In most cases, the duration of memory overload is transient, even at high memory over-committing ratios. In a data center with realistic web workload, about 88% of overload events take less than 2 minutes, where 30.6% of the overload events only last 10 seconds or shorter [40]. Thus, memory overload could be classified to two types: transient overload and sustained overload. VM live migration is suitable for sustained overload, while it is too heavyweight for transient overload. As most memory overload events are transient, a lightweight approach with more flexible use and reuse of existing hardware memory resources is preferable for handling memory overload of PMs.

Remote Direct Memory Access (RDMA) is a method of reading data from, and writing data to, the memory of a remote machine. It neither involves network software stack and kernel nor consumes any CPU time for data transfers with the remote memory [17]. In contrast, transmitting a message in traditional socket-based model requires copy operations through the kernel space. Therefore, RDMA eliminates context switch, intermediate data copies in various stack, and protocol processing from traditional model. It is one of the capabilities of InfiniBand-networked PMs. InfiniBand is a special networking technology that provides extremely high data throughput and very low latency particularly in high-performance computing, in-network computing, and data-intensive computing environments [27, 42, 43]. It is also used for reducing memory pages transfer time of the VM migration [10], and disaggregating computing resources of the cloud orchestration [2]. While RDMA and InfiniBand networking have been utilized for disaggregated memory, there is a lack of efforts on exchanging inactive memory pages to a remote PM through RDMA for handling the memory overload problem.

In this paper, a memory sharing system, which integrates a mechanism of threshold-based memory overload detection, is presented for handling memory overload of InfiniBand-networked PMs in data centers. It enables a PM with memory overload to automatically borrow memory from a remote PM with spare memory. Similar to swapping to secondary memory, i.e., disks, inactive memory pages are swapped to the remote PM with spare memory as a complement to VM live migration and other options for memory tiering, thus handling the memory overload problem. Overall, the main contributions of this paper include:

1) A memory sharing system architecture is presented for handling a memory overloaded PM in cloud data centers by utilizing spare memory resource of another PM;
2) A unified control algorithm is designed for a PM to automatically borrow memory when memory overloaded and lend spare memory when feasible; and
3) The memory sharing system is physically implemented and experimentally evaluated in terms of its functionality and read/write performance.

The measured performance of our memory sharing system in read/write speed is similar to that from accessing a local non-volatile memory express solid-state drive (NVMe SSD).

The rest of the paper is organized as follows: The section of Background and Related Work discusses background and related work. The architecture of our memory sharing system is presented in the System Architecture of Memory Sharing section. The Memory Sharing Control Algorithms section designs the control algorithm for memory borrowing and lending. This is followed by system implementation in the section of Memory Sharing System Implementation. The Experimental Evaluation section conducts experiments for performance evaluation. Finally, the paper is concluded in Conclusion section.

Background and Related Work

This section discusses the background and reviews related work on memory balancing in a single PM and memory sharing in distributed PMs. Then, it summarizes the technical gaps and motivates the work of this research. Live VM migration and traditional RDMA techniques have been briefly discussed in the section above. As they are

not directly related to the main theme of this paper, they will not be further reviewed in this section.

Over the years, the rate of resource utilization has been enhanced in cloud data centers through the virtualization-based over-commitment strategy and server consolidation technologies. However, there is still a big room for its further improvement. For example, Alibaba's large-scale cloud systems show that over-committing and under-committing problems can coexist in cloud data centers [15].

The challenge is how to further increase the memory utilization of PMs without worrying about memory overload or frequent live VM migration operations. Firstly, over-committed PMs should not be assigned extra computation tasks to a higher possibility of memory overload. Secondly, under-committed PMs are kept online, even if an over-commitment strategy and server consolidation operations are applied.

Memory Balancing in a Single PM

In general, dynamic memory balancing can be adopted to re-balance provisioned memory resources for VMs running in a single PM. Several memory management mechanisms and policies are designed to dynamically re-allocate memory resources of VMs through the memory ballooning technique. Moltó et al. have described a memory over-subscription framework for over-committing memory resource of PMs in Cloud, where ballooning technique is used for memory reallocation and VM live migration is used to prevent memory overload of the PM [29]. Waldspurger et al. have introduced the concepts of idle memory tax, content-based page sharing, and hot I/O page remapping, for VMware ESX Servers [38]. Zhang et al. have designed an automatic memory control system based on idle memory tax for Xen hypervisor [46]. However, the continuous calculation of the idle memory tax for each VM causes a high CPU overhead. In addition, memory can be balanced among VMs in a single PM through a prediction-based strategy with periodical memory monitoring, prediction, and reallocation. A memory predictor has been developed to estimate the amount of re-claimable memory and additional memory required for reducing VM paging penalty [47]. Wang et al. have studied to predict memory usage based on the working set size of VMs [39]. They have used a simulation tool to represent a sequential workload, and an offline profiling tool which can incur huge overhead if used online. Such prediction could be difficult to build in an environment with time-varying workload.

Remote Memory for PMs

Memory balancing among PMs is more complicated than among VMs in a single PM because each single PM has a fixed amount of physical memory. As the memory resource of a PM cannot be physically moved to another PM, moving memory pages between the RAM and secondary memory is implemented in operating systems to overcome the limitation of the physical memory in PMs. Extending the concept of secondary memory, the remote memory paging technique has also been developed. It enables a memory disaggregated PM to use free memory of other PMs in the same local network over a network connection [34]. Efforts have been made towards disaggregated memory for various targets, such as PMs, hypervisor, and computation.

To mitigate the memory load of a PM, partial memory resource from remote PMs can be abstracted as a block device and then used as a local swap device [1]. Choi et al. have evaluated the performance of using a remote block device as a swap through TCP/IP and RDMA [9]. Srinuan et al. have designed a mechanism of remote memory tracking and swap space management for the use of memory resource from remote PMs [36]. However, these efforts require dedicated PMs to provide remote memory resource only, implying that the CPU resources of these dedicated PMs are wasted. In addition to one backend swap device, several frameworks are designed in the form of hybrid swap. For example, the reports in [16, 25] utilize remote memory and a hard disk together for enhanced reliability. However, constant reading/writing on a disk slows down the performance of other I/O operations and consumes network bandwidth if the disk is connected via a network such as iSCSI. Cao et al. have designed a hierarchical disaggregated memory orchestration system [7]. The system enables inactive page compression and swap-out optimization for a reduced size of page. Also, it is embedded with a selection mechanism to choose the most suitable back-end device. The hybrid swap architecture is further enhanced by Newhall et al. through allowing node RAM, disk, flash SSD, PCM, and network storage devices as a swap device [32]. All these approaches not only have I/O performance impact, but also demand other resources and consequently increase extra operation costs. For example, writing inactive page to local device, such as disk and SSD, can impact I/O performance, meanwhile choosing most suitable back-end device can produce computation overhead. Moreover, Oura et al. have designed multi-thread page-swap protocols to accelerate remote memory enabled swap operations [33]. However, the design is for large memory and requires modifications to user programs, increasing the difficulty to end users.

Remote memory paging works not only on multiple PMs, but also on multiple VMs on a single PM. Koh et al. [23] and Lim et al. [26] designed hypervisor-oriented frameworks to swap inactive memory pages from VMs

to remote memory connected to the host PM. Cao et al. [6] further added page compression during memory paging from a VM to the host PM. Kocharyan et al. [22] presented a system to monitor the working sets of VMs and re-claim unused memory of VMs through Xen balloon driver for paging memory from memory overloaded VMs to VMs on other PMs. Another implementation based on the memory management of the hypervisor was MemX [12, 20]. In addition, Williams et al. indicated that using remote memory for VMs requiring extra memory for a sustained period of time was an inefficient way [41]. For such VMs, they prefer automatic and live VM migration, which is a different topic from what we are discussing in this paper. Overall, a single PM usually has multiple running VMs, and managing a remote swap device for VMs is more complicated than for PMs.

It is worth indicating that swapping inactive memory pages may not be the most efficient method for some memory intensive computations [21]. Distributed in-memory key-value store can be used for applying remote memory resource on a program [13]. The memory load of a PM can be mitigated if an application stores its runtime data in remote memory. SpongeFile [14] is a logical byte array comprised of large chunks that can be stored in other PMs. It is designed specifically for distributed MapReduce computation. Another implementation is to leverage remote memory and RDMA for accelerating relational database management systems when there is insufficient memory, rather than using slower media such as SSDs or HDDs, which can significantly degrade workload performance [24]. However, this and other similar methods are designed for special purposes, and thus are not a general solution to the mitigation of the memory load of PMs in cloud data centers.

Commercial Alternative Solutions

Commercial alternative solutions tend to share more than just memory resources. CPU cores, memory, and I/O of multiple PMs can be aggregated to present as a single operating system, which involves a wide variety of techniques, from custom hardware and distributed hypervisors to specialized operating system kernels and user-level tools [19]. ScaleMP vSMP aggregates multiple commodity x86-based PMs, such as a classic cluster, into a single virtualized system [35]. It runs a patched Linux operating system and uses InfiniBand-enabled network for interconnecting each PM [44]. TidalScale is another commercial software solution to aggregate computing resources of multiple PMs, which allows hot-swap individual PM for upgrade or repair while the mission-critical workload continues to run [4, 31]. In contrast to ScaleMP vSMP, TidalScale does not require changes to either the application or the operating system. In addition, custom hardware for memory sharing relies on emerging open standards, such as Open Memory Interface (OMI) and Compute Express Link (CXL). IBM's Power10 has a memory clustering feature called memory inception, which allows one PM to map its address space to the physical memory of another PM by leveraging OMI memory [37]. It allows a PM to run large memory workloads that go beyond its capacity by borrowing memory from other PMs. A similar design is memory expansion and pooling. The CXL protocol allows the CPU to share and address memory in the attached CXL device [28]. It can serve as a system-expansion interconnect, which runs on the Peripheral Component Interconnect Express (PCIe) bus with protocol-layer enhancements, that make it especially apt for memory attachment [28]. It gives memory controllers the capability of implementing memory pooling, where a large amount of memory can be segmented into various regions which are then allocated to different PMs [11]. However, these solutions require specialized hardware and operating systems which prevent memory sharing in most existing data centers.

Technical Gaps and Motivation

Memory balancing and remote memory are two major candidate approaches for handling memory overload of PMs in cloud data centers. However, memory balancing requires the coexistence of over- and under-committed computation tasks in the same PM. The total memory requested from all VMs on the PM are capped by the memory capacity of the PM. In comparison, remote memory for PMs does not have such a limitation. However, existing efforts in remote memory for PMs require a pre-configuration of disaggregated memory on dedicated PMs. Such efforts cannot help cloud data centers to increase the memory utilization of PMs since they are unable to take advantage of utilizing spare memory resource of running PMs. Furthermore, multiple or hybrid disaggregated memory designs are not essential for handling memory overload for PMs in cloud data centers.

Therefore, technical gaps exist towards our memory sharing requirements for PMs in cloud data centers, i.e., 1) a PM can access remote memory resources only when it becomes memory overloaded; and 2) when a PM has spare memory resource, it can dynamically share out its memory resource to a remote PM. 3) memory sharing should be transparent to applications and end users, and can be deployed in traditional uniform amd64 PM data centers since memory overload will not appear if the cloud data centers apply modern emerging technologies to aggregate and virtualize physical resources, such as IBM Power10. This motivates the research and development of this paper on memory sharing for handling memory overload on PMs in cloud data centers.

System Architecture of Memory Sharing

This section describes our memory sharing system architecture. Depending on the memory utilization, a PM may require additional memory from other PMs, or has spare memory to share out to other PMs. Therefore, each PM will play a role at a time as either a memory borrower or a memory lender. A PM that is using remote memory resources for handling memory overload is a borrower PM. In contrast, a PM that is providing its memory resource for a remote PM to use is referred to as a lender PM. In general, a PM may change its role from a borrower to lender or vice versa over the time.

Our memory sharing system is shown in Fig. 1. It is composed of three main components: a controller, an elastic block device, and a data transfer protocol:

- The controller is implemented as a user space application. It decides when remote memory is required.
- The elastic block device is embedded in a kernel space module. It stores inactive memory pages exchanged from memory device.
- The data transfer protocol is designed for commands execution and data transfer. It defines how memory pages are transferred between the borrower and lender PMs.

As shown in Fig. 1, the memory sharing requires a cooperation within the user space application (Controller), the kernel space module (Elastic Block Device), and the data transfer protocol. The solid arrows in the figure represent program commands execution, while the dotted line arrows indicate data transfer routes. The hardware directly involved in the system operation includes the memory on each PM and InfiniBand-internetworked PMs.

Controller

A controller is designed for each PM that participates in memory sharing. It monitors the system memory information and manages the elastic block device of the PM. Logically, a PM may borrow memory from, or lend memory to, another PM. Therefore, the controller on the PM acts as either a borrower controller or a lender controller, but not both, at a time. If a PM is required to borrow memory from a remote PM that has spare memory to share out, the controller on this PM becomes a borrower controller, while the controller on the remote PM becomes a lender controller. With the change of the memory status of the PM, the controller on a PM may change its role from a borrower controller to a lender controller or vice versa over time.

For a borrower controller running on a borrower PM, if the used memory of the PM reaches a given threshold, the borrower controller communicates and exchanges memory sharing information with a lender PM for establishing memory sharing, followed by activating the elastic block device, which is linked to a lender PM through data transfer protocol. Then, it enables swap space so that the kernel exchanges inactive memory page from the physical memory to the elastic block device. If the borrower PM is no longer considered as being overloaded, the borrower controller disables swap and detaches the elastic block device of remote memory.

Lender controller is activated passively by memory sharing request. When handling a request of sharing memory from a borrower PM, it creates an elastic block device and reserves sufficient memory resources for sharing. If the elastic block device is disconnected from the borrower PM, the lender controller will clean up the elastic block device and free up the memory space that was shared out previously.

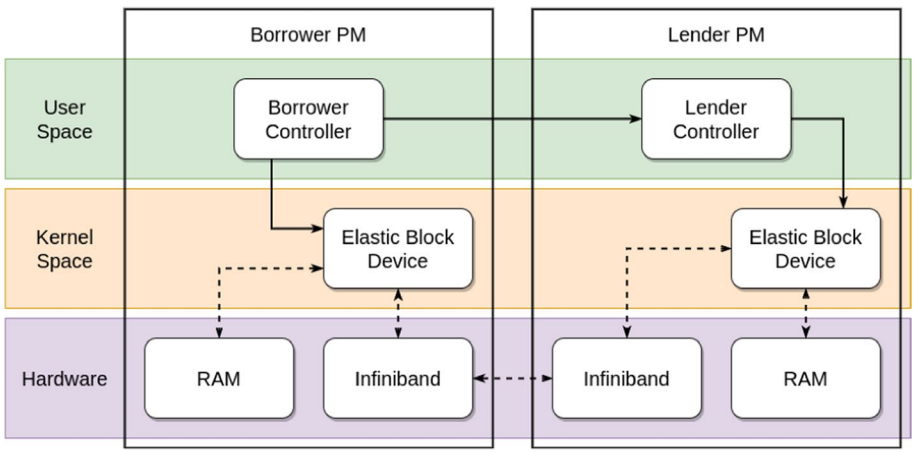


Fig. 1 System architecture of memory sharing

Elastic Block Device

The elastic block device is a logical storage interface on a borrower PM or a physical storage disk on a lender PM. While logically it is used like a hard disk, it behaves differently on borrower and lender PMs.

On a borrower PM, the elastic block device abstracts a hard disk to the operating system. It performs as a logical interface to accept block I/O requests. These block I/O requests are then passed through to a lender PM via InfiniBand networking.

On a lender PM, the elastic block device abstracts an in-memory hard disk that stores sectors in RAM. As memory is allocated only when requested, the amount of memory the elastic block device occupies changes over time depending on how much data needs to be stored. For example, on an elastic block device of 2 GiB, only 1 GiB memory will be consumed if only 1 GiB data needs to be stored. When the stored data is deleted from the elastic block device, the elastic block device will free up this previously occupied block of memory.

Data Transfer Protocol

The data transfer protocol for memory sharing is developed based on NVMe over Fabric (NoF) from existing systems. NVMe is designed to work over a PCIe bus. Legacy storage stacks for accessing a storage device over the network could be used to operate NVMe devices. However, the requirements of synchronizations and command translation largely offset the benefits of NVMe devices for remote access.

NoF is a protocol for transferring NVMe storage commands between client nodes and target nodes over InfiniBand or Ethernet networks through RDMA [18]. It standardizes the wired data transfer process and hardware drivers for efficient access over RDMA-capable networks with minimal processing required by the target node.

The data transfer protocol together with the elastic block device in our memory sharing system is shown in the logical diagram of Fig. 2. It encapsulates a block I/O request, sends the request to the target node through RDMA, decapsulates the block I/O request, and finally passes the block I/O request to the storage elastic block device. This enables efficient data transfer from a borrower PM to a lender PM and vice versa.

State Machine

With the design of the dual controller role described previously in the section of Controller, the controller has three states: borrower, lender, and neutral. It stays

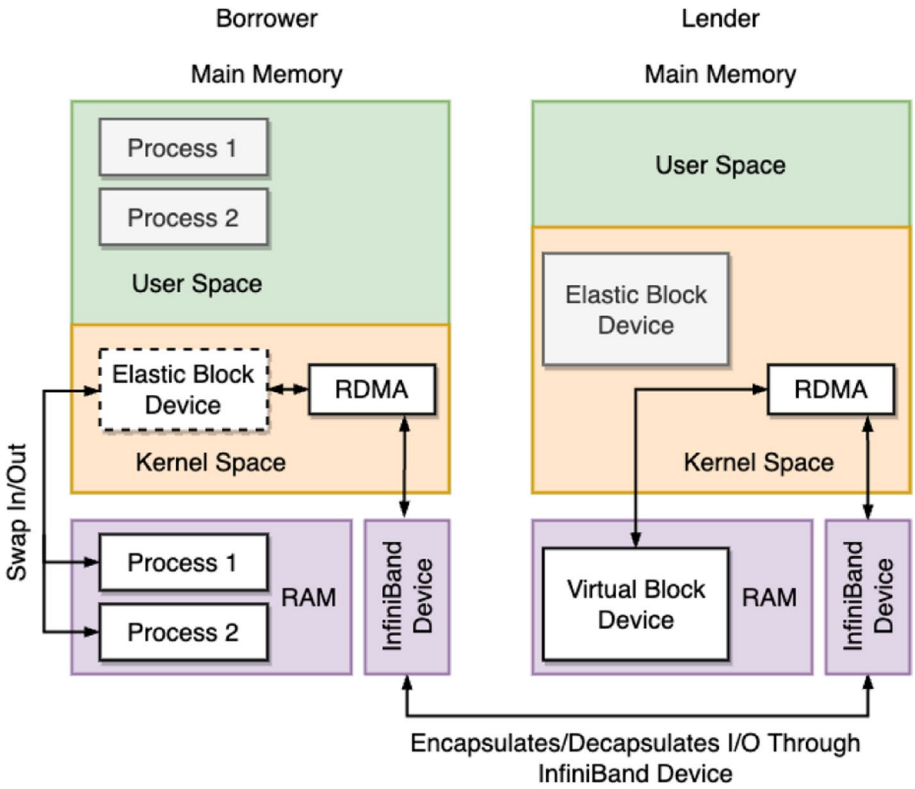


Fig. 2 System swap diagram in our memory sharing system

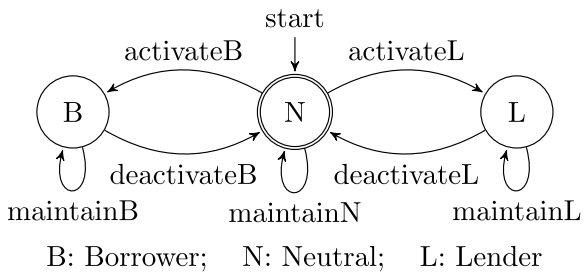


Fig. 3 State machine of the memory sharing system

at the Neutral state when the PM neither borrows nor lends memory resource. Transitions of these three states are demonstrated in state machine shown in Fig. 3. They are discussed below.

When a PM boots up, it runs under a low level of workload. The controller stays at the Neutral state with no need to borrow memory from other PMs. Meanwhile, the PM has not shared out any spare memory resource yet to other PMs.

Then, the controller may stay at the Neutral state if it does not need to borrow memory from other PMs or nowhere to borrow memory. The controller may transit from the Neutral state to the Borrower state if it requires more memory from other PMs, or the Lender state if it is requested for memory from other PMs and has spare memory to share out.

For the Borrower state, there are only two possible transitions: to either itself or back to the Neutral state. If memory borrowing is maintained, the controller stays at the Borrower state. Otherwise, if memory borrowing is no longer needed or the lending PM cancels its memory sharing, the controller deactivates the memory borrowing and switches back to the Neutral state.

Similar to the Borrower state, the Lender state also has two possible transitions: to either itself or back to the Neutral state. If the memory lending is maintained, the controller stays at the Lender state. Otherwise, if memory lending becomes infeasible due to the lack of spare memory or the borrowing PM terminates its memory borrowing, the lender controller deactivates memory lending and transits back to the Neutral state.

In the state machine of Fig. 3, the transitions of the states occur periodically in each period rather than instantly. They require the operations of all components in the system architecture on same PM. They also need the cooperation of the controllers on both borrower PM and lender PM for memory sharing between to the two PMs. This will be achieved through the design of signalling as will be discussed later. Moreover, it is seen from the state machine Fig. 3) that there are no direct transitions between the Borrower state and Lender state.

Signaling of Events

For cooperation of various components in a controller on a single PM or two controllers on two PMs, signaling of events becomes important in architecture design of the memory sharing system. In terms of the intended functions, there are generally two types of signaling: signaling for memory sharing activation and signaling for memory sharing deactivation.

There are two scenarios of memory sharing activation, as shown in Fig. 4. Normally, lender PM-B has to prepare its lender elastic block device and change its state to lender before creating borrower's elastic block device and establishing memory sharing connection on borrower PM-A (Fig. 4a). However, lender PM-B may reject

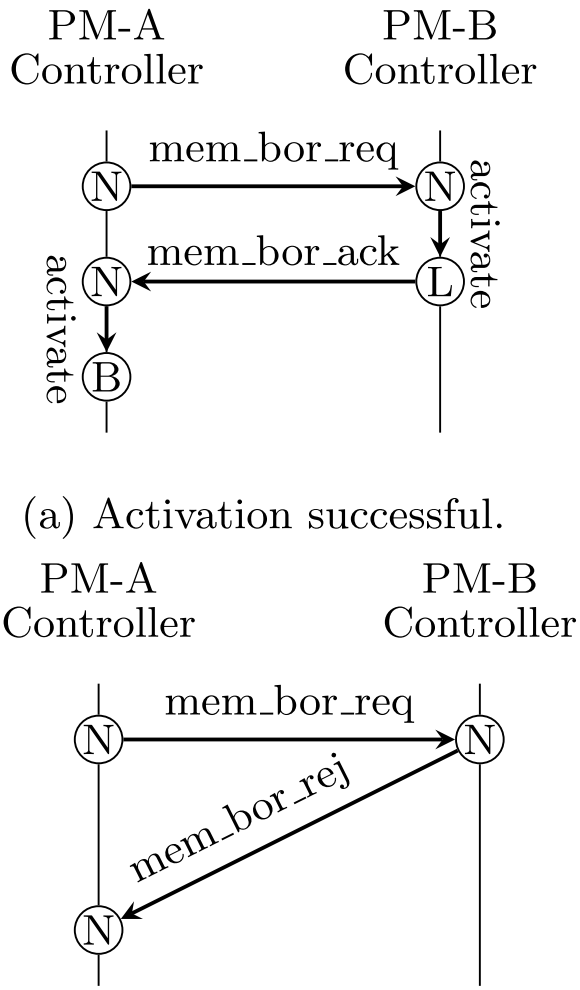


Fig. 4 Signaling for memory activation

memory sharing if it does not have enough spare memory resource for lending (Fig. 4b).

Memory deactivation can be triggered by either borrower PM-A or lender PM-B. If memory overload is no longer existed, borrower PM-A will disconnect remote memory resource before notifying lender PM-B for cleanup of lender elastic block device and reverting the controller state (Fig. 5a). In a similar way, if lender PM-B is likely becoming memory overloaded, i.e., it no longer has spare memory for sharing, it will ask borrower PM-A to stop borrowing memory (Fig. 5b). During the remote memory deactivation, PM-A has to reduce its memory usage followed by pulling back data already stored in remote memory. Reducing memory usage can be done by migrating workloads to other PMs, or terminating workloads with low priority.

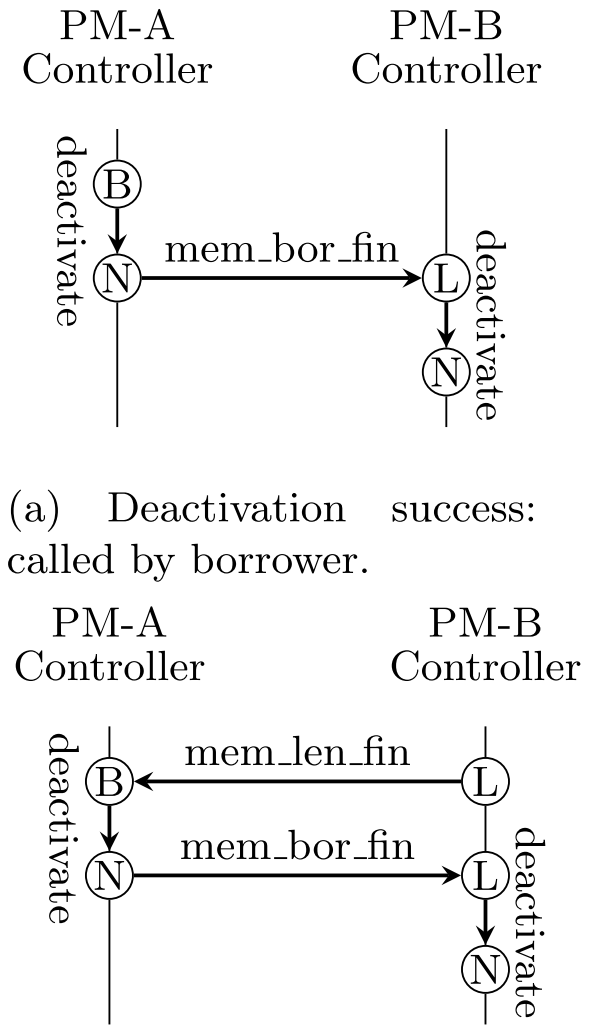


Fig. 5 Signaling for memory deactivation

Memory Sharing Control Algorithms

This section describes how the controller on a borrower PM determines to activate memory sharing and how the controller on a lender PM responds. The pseudo code of controller algorithm is shown in Algorithm 1. In addition, notations and symbols used in this paper are listed in Table 1.

```
   Initialization: Set StateFlag ← Neutral
1  for Every monitoring period do
2      if StateFlag is Neutral then
3          Execute Algorithm 2 for Neural state;
4      else if StateFlag is Borrower then
5          Execute Algorithm 3 for Borrower state;
6      else
           // StateFlag is Lender
7          Execute Algorithm 4 for Lender state;
8  return
```

Algorithm 1 Controller AlgorithmA PM needs to have the ability to dynamically switch its role between a borrower PM and a lender PM, as well as keep Neutral state. Algorithm 1 describes an unified flag-based controller for all memory sharing operations. In each period cycle, controller decides next state flag depends on its memory utilization and memory sharing request.

There are three flags, Neutral, Borrower, and Lender, which correspond with three states described in the section of State Machine. Controller with Neutral flag decides if the PM can maintain Neutral flag or needs transition to other flag: either borrower controller or lender controller flag (lines 2-3). For Borrower and

Table 1 Symbols and notations

M_b	Memory used by kernel buffers
M_c	Memory used by page cache and slabs
M_f	Memory free
M_r	Memory reserved for other applications
M_{req}	Memory requested by borrower PM
M_s	Memory shareable
M_t	Memory total
M_u	Memory used
S_t	Swap total
S_u	Swap used
T_{lower}	Lower threshold to trigger memory overload detection
T_{upper}	Upper threshold to trigger memory overload detection

Lender state, controller decides if the PM can maintain current state or needs revert back to Neutral state.

Activation or Deactivation of memory borrower and lender happens on state transition. When transiting from neutral to other states, memory borrower or lender is activated. On the other hand, memory borrower or lender is deactivated when transiting to Neutral. This is decided by controller with borrower flag (lines 4-5) and lender flag (lines 6-7). Hence, Borrower and Lender state must be transited to Neutral state in prior to switch to lender or borrower.

Algorithm for Neutral State

Controller algorithm for Neutral state is shown in Algorithm 2. Default flag (maintain Neutral state) will be returned if there is no API request received or requested remote memory resource cannot be fulfilled (lines 1-5). During stay of Neutral, current memory usage is examined against threshold in order to detect memory overload (lines 4-5). A request of borrowing memory resource *mem_bor_req* is sent if the threshold is reached.

```
   Data      : StateFlag, M_req, M_s, M_t, M_u, S_t, T_upper
   Output    : StateFlag
   Initialization: Set StateFlag ← Neutral
1  if Transition to Neutral itself, i.e., no request or request
     infeasible (M_req > M_s), then
2      if request infeasible, i.e., M_req > M_s, then
3          Reject it by sending response mem_bor_rej;
4      if memory borrowing needed, i.e.,
         M_u > T_upper × M_t, then
5          Send memory request mem_bor_req;
6  else if Transition to Borrower state, i.e., mem_bor_ack
     received, then
7      Activate memory Borrower;
8      StateFlag ← Borrower;
9  else
       // Transiiton to Lender state
10     Activate memory Lender;
11     Send response mem_bor_ack;
12     StateFlag ← Lender;
13 return StateFlag;
```

Algorithm 2 Controller Algorithm for Neutral StateState transitions are described in lines 6-12. Receiving *mem_bor_ack* results in memory borrower activation (line 6-8). Memory lender is activated only if requested memory resource m_{req} is smaller than memory reserved for sharing m_s (lines 9-12). Otherwise, borrower's request is rejected (lines 2-3). Calculation of m_s will be explained in Section Algorithm for Lender Controller.

Algorithm for Borrower Controller

Activation and deactivation of borrower controller is determined by a dual threshold mechanism in order to avoid faulty or too frequent activation of memory sharing. The first threshold is an alarm threshold T_{upper}. In our memory sharing system, a PM is considered as memory overloaded if its memory usage is beyond alarm threshold T_{upper}. As realistic memory usage may fluctuate around T_{upper}, another threshold T_{lower} is set to decide whether memory sharing can be deactivated.

```
   Data      : StateFlag, M_u, M_t, S_u, T_lower
   Output    : StateFlag
   Initialization: Set StateFlag ← Borrower
1  if memory borrowing still needed (i.e.,
     M_u + S_u >= T_lower × M_t) and feasible (i.e., no
     mem_len_fin received) then
2      Do nothing (i.e., maintain Borrower stater);
3  else
4      Deactivate memory borrower;
5      Send mem_bor_fin;
6      StateFlag ← Neutral;
7  return StateFlag;
```

Algorithm 3 Controller Algorithm for Borrower State

```
   Data      : M_t, M_u, StateFlag, T_upper
   Output    : StateFlag
   Initialization: Set StateFlag ← Lender
1  if memory lending not terminated by Borrower, i.e., no
     mem_bor_fin received, then
       // Maintain Lender state
2      if lending likely becomes infeasible
         (M_u > T_upper × M_t) then
3          Send mem_len_fin to prepare deactivation;
4  else
5      Deactivate memory Lender;
6      StateFlag ← Neutral;
7  return StateFlag;
```

Algorithm 4 Controller Algorithm for Lender StateAlgorithm for borrower controller decides whether memory borrower should be maintained. It only involves the second threshold part of the dual threshold mechanism, while the first threshold part is involved in line 4 of Algorithm 2. Borrower controller is maintained if memory usage is still higher than threshold T_{lower} and no termination request *mem_len_fin* is received (line 1).

Deactivation of memory borrower requires sum of used memory M_u and used swap space S_u be smaller than threshold T_{lower} because memory pages in swap space will be fetched back to main memory during deactivation of memory borrower. In addition, request *mem_lend_fin* may be sent from lender PM to indicate that lender PM is unable to keep sharing memory remotely. After deactivation of memory borrower, it is necessary to notify memory lender by sending a *mem_bor_fin* request.

Algorithm for Lender Controller

This section describes how to calculate size of shareable memory resource and when memory lender can be deactivated.

Remote memory resource can be reserved via a pre-configuration or dynamical configuration. The capacity of the shareable memory resource, M_s, can be calculated from either a pre-configured method or a dynamic method. The pre-configured method derives M_s from the total memory M_t less the reserved memory for all other applications, M_r, i.e.,

$$M_s = M_t - M_r \tag{1}$$

The dynamical configuration requires M_u to be known in advance. The dynamic method calculates M_s from the total memory M_t less used memory M_u, i.e.,

$$M_s = M_t - M_u \tag{2}$$

With this method, M_s needs to be updated periodically with Eq. (2) because M_u changes over time.

The shareable memory resource M_s is examined against requested memory resource in order to determine whether lender PM has sufficient spare memory resource for sharing. It is different from how much memory has been shared out. When the lender PM shares its memory resource to a borrower PM, it only reserves and shares out requested amount of memory resource sent by the borrower PM.

Deactivation of memory lender depends on whether request *mem_bor_fin* is received (line 1), as shown in Algorithm 4. Request *mem_bor_fin* sent by borrower PM triggers deactivation of memory lender. However, the controller checks whether the lender PM becomes memory overloaded in the process of maintaining the Lender state (line 2). If the lender PM is likely becoming memory overloaded, the lender controller will send a message *mem_len_fin* to the borrower PM for the termination of the memory sharing.

Memory Sharing System Implementation

All components of the proposed memory sharing system are deployed on each of the PMs in a cloud data center though not all of them need to execute at the same time. Their implementations are shown in Fig. 6 from the programming perspective. Overall, the controller of our memory sharing system is implemented as a user space program because it requires communications between PMs. Network communications must be managed by a kernel based netfilter. The elastic block device is implemented as a kernel module interacted with other kernel functions. Moreover, our memory sharing system is implemented in C and Go languages.

Controller Implementation

The controller is implemented in a modular program with several bundled modules including a memory usage

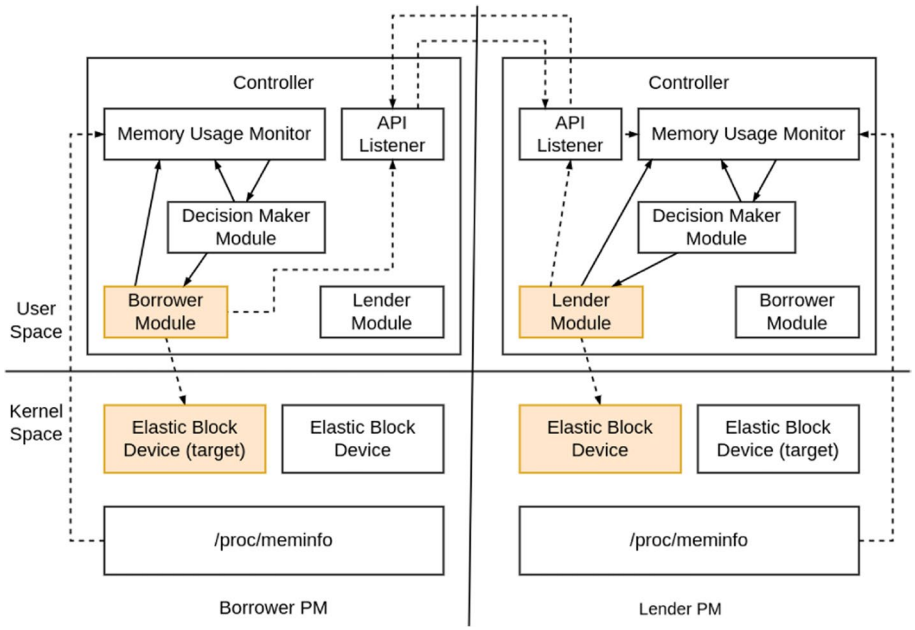


Fig. 6 Implementation of memory sharing in user and kernel spaces on each PM. Solid arrows represent state transition between four modules: memory usage monitor, decision maker module, borrower module, and lender module. Dashed arrows represent input for triggering state transition

monitor, an API listener, a decision maker module, a borrower module, and a lender module. The memory usage monitor and API listener run concurrently in separate lightweight threads managed by the Go runtime.

They dynamically activate/deactivate the borrower module or lender module, as shown in the orange blocks in Fig. 6. Solid arrow represents state transition between four modules: memory usage monitor, decision maker module, borrower module, and lender module. Dashed arrow represents input for triggering state transition. After collecting memory information, state is transited from memory usage monitor to decision maker module for deciding if borrower or lender needs to be activated or do nothing. Such logic is presented as flowchart in Fig. 7, where memory usage monitor keeps periodically gathering memory information until controller exits. Moreover, activation of lender module depends on event from API listener. It is activated only if decision maker module finds lender PM can fulfill requirement of memory sharing request sent by borrower PM.

More detailed operations of the controller and the relationships of the controller modules and Elastic Block Device are depicted in a system logic diagram of Fig. 8. Red line arrow represents final action of each system component. Apart from the controller, implementation of other system components will be described in subsections below.

Memory Usage Monitor. This module tracks system memory usage and passing information to decision maker module. It obtains the memory information of the

Fig. 7 Controller Flowchart

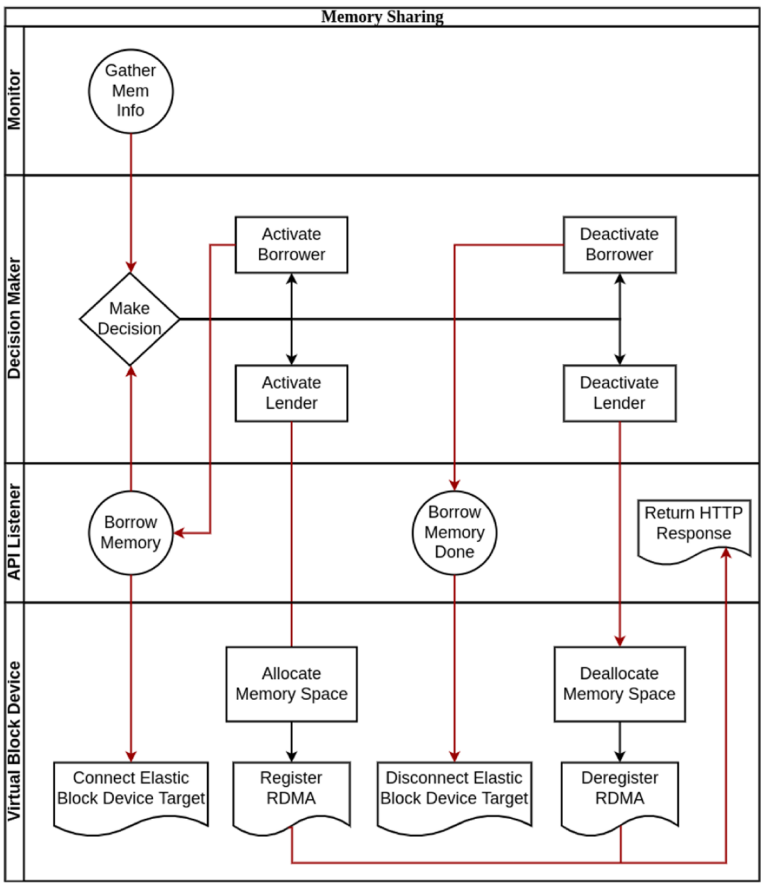


Fig. 8 System logic in swim-lane diagram

PM periodically via a low-level system call to *kernel/proc/meminfo*.

API Listener. This module listens on controller APIs which are served and requested through HTTP methods. There are five API methods: *mem_bor_req* and *mem_bor_fin* are sent by borrower PM; *mem_bor_ack*, *mem_bor_rej*, and *mem_len_fin* are sent by lender PM.

Decision Maker Module. The dual-threshold mechanism is implemented in this module to decide if borrower module or lender module needs to be activated. Activation of borrower module is put into account if passed memory information comes from memory usage monitor only, while activation decision for lender module is made if passed memory information additionally contains requested memory resource M_{req} received from API listener.

Borrower Module. The commands *mem_bor_req* and *mem_bor_fin* are sent out by the borrower module for borrow operations of remote memory resource. Command *mem_bor_req* contains a parameter M_{req}, which defines how much remote memory resource is required. Lender's return (*mem_bor_ack*) has a parameter NoF Qualified Name (NQN), which is used to identify the elastic block device on the lender PM. Once NQN is obtained from the lender PM, the borrower module will use the NVMe-cli tool with NQN to connect the remote elastic block device, and then set up and enable the swap device. If memory overload no longer exists, the borrower module will remove the swap device and then issue a *mem_bor_fin* command to notify lender PM. In addition, if *mem_len_fin* is received, the borrower module will move remote paging to local swap device and then issue a *mem_bor_fin* command.

Lender Module. When this module is activated, it configures the elastic block device with the size of remote memory, sets the elastic block device address and NQN for NoF subsystem, and returns *mem_bor_ack* with NQN. *mem_bor_rej* may be returned for unable to share memory. In response to *mem_bor_fin*, the lender module asks the elastic block device module to clean up the memory previously shared out. In addition, the lender module communicates with the elastic block device through Netlink (*AF_NETLINK*), which consists of a standard sockets-based interface for user space processes

and internal kernel APIs for kernel modules. Moreover, if lender PM becomes memory overloaded, lender module will send a *mem_len_fin* command to borrower PM for requesting halt of memory sharing.

Implementation of Elastic Block Device

In the implementation of our memory sharing system, the elastic block device as a logical interface on a borrower PM is generated by NoF. On a lender PM, the elastic block device is implemented as a temporary physical storage. Its logical structure is shown in Fig. 9. The elastic block device receives commands from user space for creating and discarding elastic block device on demand. In Fig. 9, the yellow-shaded blocks represent three main data structures of the elastic block device: *netlink_kernel_cfg* (netlink), *gendisk*, and *radix_tree_root* (radix tree). They are discussed below.

Data structure netlink_kernel_cfg (netlink). Netlink is served for listening requests from the lender module. The request in a netlink message is to either configure gendisk (*set_capacity()*, *add_disk()*, etc.) or clean up the radix tree (*radix_tree_delete()*).

Data structure gendisk. A block device is defined by gendisk, which describes how to handle block I/O requests. *block_device_operation* only needs the following structure data of the block device: heads, sectors, and cylinders. An alternative *make_request()* function for the block device is needed to define for *request_queue*. *make_request()* does data read and write operations through calling *radix_tree_lookup()* and *radix_tree_insert()*, respectively.

Data structure radix_tree_root. Blocks are stored in data structure of radix tree provided by Linux kernel library < *linux/radix-tree.h* >.

Experimental Evaluation

This section conducts experiments to verify the feasibility of the memory sharing system presented in this paper for handling memory overload. It will show that remote memory can be attached and detached dynamically on the fly. It will also demonstrate that the read/write performance of remote memory operations is acceptable.

Feature Comparisons with Existing Solutions

There are two essential feature requirements for memory sharing in cloud data centers. Firstly, memory sharing should be transparent to process, which allow a PM or a running process uses remote memory resource without modification on program source code. Otherwise, virtualization environment, such as VM and container, cannot be benefited by memory sharing. Secondly, a PM should be able to dynamically share out its memory resource or

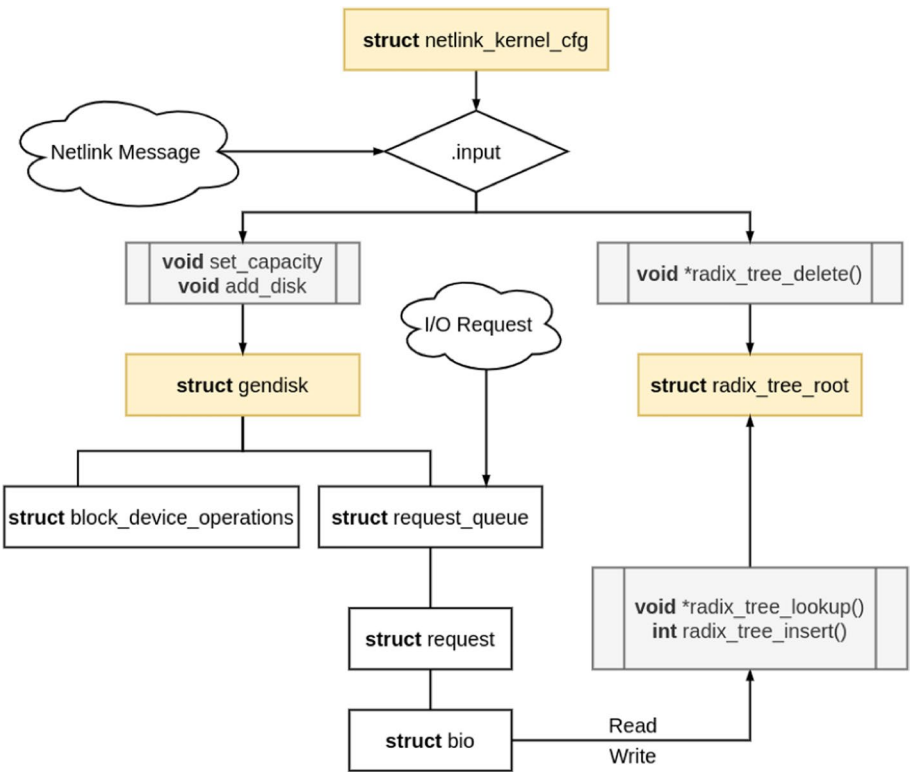


Fig. 9 The kernel module of an elastic block device as a storage on a lender PM

Table 2 Comparisons of our method in this paper with existing solutions

	Elastic block device based	Transparent to process	Dynamic PM role
AIFM [34]			
DLM [33]			
COMEX [36]		✓	
XMemPod [7]		✓	
SMB [1]	✓		
RMBD [9]	✓	✓	
Infiniswap [16]	✓	✓	
HPBD [25]	✓	✓	
Nswap2L [32]	✓	✓	
Our Framework	✓	✓	✓

use remote memory resource because any PM in cloud data centers can be memory overloaded or have spare memory resource.

Feature comparison with existing solutions is conducted based on how the solution is implemented, and two requirements described above. Result is shown in Table 2. Most of solutions are elastic block device based and are transparent to process by design. However, all of them are statically memory sharing, which pre-configures both the PM for sharing out memory and the PM for accessing memory resource on a remote PM. In cloud data centers, memory overload problem cannot be handled by these memory sharing solutions due to lack of the ability to dynamically share the spare memory of any PM. On the other hand, our framework enables the ability of dynamic memory sharing on demand by introducing the dynamic PM role.

It is worth mentioning that the memory sharing presented in this work is not designed to replace VM live migration. Rather, it will effectively reduce the number of live VM migrations induced by memory overload. This means that the memory sharing presented in this paper complements existing VM live migration. The memory sharing is suitable for the management of transient memory overload that lasts for a short period of time. It can be pre-activated or activated instantly. However, for sustained memory overload, memory sharing is not recommended and VM live migration will be more suitable. After successful migration, local memory will be used instead of remote memory. It is noted that live migration requires to transfer the memory pages of the VM from one PM to another PM and will typically cause unavailability of the VM during the migration. Depending on the size of the VM, live VM migration may take seconds or longer. It may even use RDMA of InfiniBand for the required transmission.

Memory Sharing Behavior

We monitor memory information changes from start to end of PM overloading. A mock application is implemented and deployed to overload the PM with capacity of 256 GiB memory. It keeps requesting memory allocation and filling up the allocated memory with random characters, until 1.5 times of memory resource capacity is used. Figure 10 exhibits lender and borrower system runtime, where y axis represents the size of memory usage and x axis represents time. The memory borrowing procedure is triggered at around 20 seconds, at which swap is enabled for utilization. The memory usage growth from Fig. 9b confirms the usage of remote memory in the lender system. The growth trend of swap in the borrower PM (i.e., the pink area in Fig. 9a) is the same as the growth trend of RAM in the lender PM (i.e., the blue area in Fig. 9b). This means that the memory shortfall on the borrower PM is filled up by the same amount of RAM from the lender PM though our memory sharing system.

PM roles changes after 2 minutes of this experiment. Borrower PM no longer has memory overload at 124 second, while lender PM is pressed by the mock application

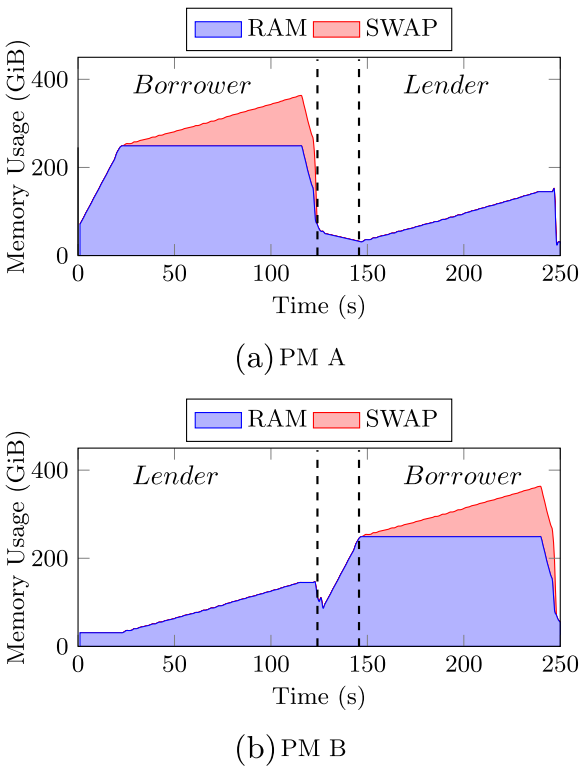


Fig. 10 Experimental results of memory usage over time. The experiments are conducted by running a mock application to overload memory resource of PM A and PM B sequentially. The memory shortfall (swap) on the PM with Borrower role is filled up by the same amount of RAM from the PM with Lender role

for overloading its memory. Lender PM, shown in Fig. 9b, becomes memory overloaded and starts borrowing memory at 147 second. It borrows memory resource from PM of Fig. 9a. At this time, RM roles of lender and borrower are switched.

Benchmark Applications

To evaluate the performance of our memory sharing system, we adopt two types of benchmarks: IOzone and DaCapo. These benchmarks are described in the following:

Benchmark IOzone. IOzone (version 3.490) is a filesystem benchmark tool, which generates and measures a variety of file operations [8]. In our tests, we select 4 KiB buffer random read and write operations for gaining theoretical performance on exchanging inactive memory pages.

Benchmark DaCapo. DaCapo (version 9.12) [5] is a Java benchmark suite used to measure the performance of memory management and computer architecture communities. It consists of a set of 14 open-source, real-world applications with non-trivial memory loads. For our tests, we select three memory-intensive applications from DaCapo applications, i.e., h2, tradebeans, and tradesoap.

Experimental Setup for Performance Evaluation

In our experiments, two InfiniBand-networked workstations are used: one as a borrower PM and the other as a lender PM. Each of them has a 24-core Intel Xeon 5118 processor, 256 GiB 2666MHz Hynix Memory, Samsung NVMe SSD PM981 (270,000 IOPS Max on 4 KiB random read, 420,000 IOPS Max on 4 KiB random write), and Mellanox ConnectX-3 Pro 56GbE adapter card that supports RDMA. The OS used is Red Hat Enterprise Linux 7.7 64-bit, whose kernel is Linux 3.10.0.

Setup for theoretical performance experiments and practical experiments are detailed in Table 3, where remote represents using our memory sharing system, NVMe represents using local NVMe SSD as swap device, and disabled represents using system memory only. IOzone is used to test theoretical performance, while three applications from DaCapo suite are used to test practical performance on two popular virtualization environment, Docker and KVM-based VM.

Table 3 Experimental setup for performance evaluation

	Baremetal	Docker	KVM/QEMU VM
Application	IOzone	DaCapo (tradesoap, tradebeans, h2)	
Swap Source	Remote	NVMe, Disabled, Remote	

The three applications selected from the DaCapo benchmark applications are executed in container and VM environments, respectively. Thus, Docker (version 19.03.8) and Kernel-based Virtual Machine (KVM) are deployed in the two workstations. The performance of our memory sharing system is measured for both Docker-based and KVM-based system configurations. Moreover, performance evaluation is conducted by comparing our memory sharing system with local NVMe SSD based swap device and local RAM only. The local NVMe SSD, acting as the fast physical storage device, represents the best case of performance in our experiment. It needs to be noticed that the local NVMe SSD based swap device is not feasible in the cloud data center because physical storage devices are located dedicatedly and connected to PMs through the network.

Docker is utilized for emulating memory-critical scenarios where available memory resources are limited to each application and thus the application is forced to use remote memory in most of its running time. The Docker image is Red Hat Universal Base Images (version 8.1), which involves Java SE Runtime Environment (build 14). Each container instance can access 512 MiB memory and all PM swap space.

KVM is utilized for emulating memory-flexible scenarios in a VM-based cloud environment, which in our design will use remote memory only when the hosting PM becomes memory-overloaded. Two VMs are deployed on a PM, where one VM runs benchmark suite applications, and the other runs mock applications that will overload the PM. The guest OS of the VMs is Fedora 33 64-bit, whose kernel is Linux 5.10.19. It involves OpenJDK Runtime Environment 20.9 (build 15.0.2). Each of the VMs is assigned 12 CPUs and 250 GiB memory because each KVM-based VM functions as a Linux process where the Linux kernel of the host PM allocates memory only when requested.

Theoretical Performance

We aim to verify that memory sharing could have reasonable performance by comparing the performance of remote memory with the fastest possible local storage device under the simulation of memory page swapping. In our experiments, remote memory is manually attached as elastic block device. 4 KiB buffer random read and write operations are tested on the attached block device. Figure 11 shows the throughout and latency results of 4 KiB buffer random read and write operations.

Our experiments show that remote memory always outperform local NVMe SSD in both throughput and latency. In comparison with RAM, remote memory has slightly

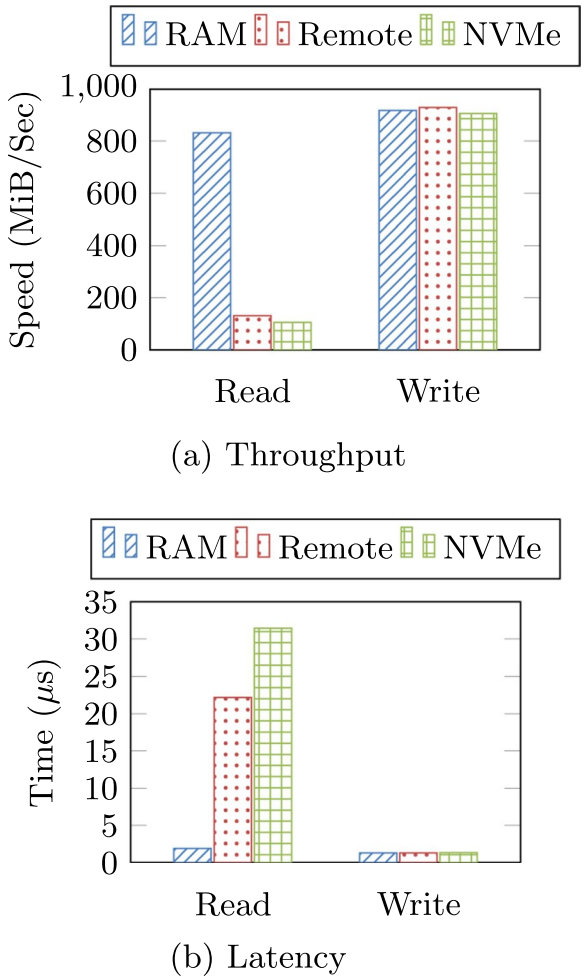


Fig. 11 The performance of random read and write operations on the 4K-buffer block I/O benchmark

higher speed than RAM on random write while is much slower than RAM on random read. This is also confirmed on operation latency. Random read on remote memory is slow because read operations end when a buffer has been put in a given destination memory address. However, write operations on remote memory are marked as being completed when buffer is sitting in the RDMA working queue for directly writing to remote memory.

Memory Sharing on DaCapo Benchmark Suite

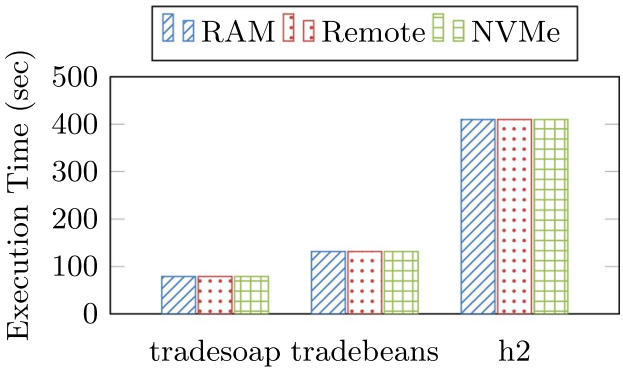
Three applications, tradesoap, tradebeans, and h2 are selected from DaCapo benchmark suite to examine performance of remote memory. It is expected that remote memory could achieve similar performance as local NVMe SSD which is currently one of fastest storage devices available.

Figure 12 shows results of running selected applications without memory overload. In this case, neither remote memory nor NVMe SSD based swap space is used because host system is not yet out of memory. Thus, remote memory is not triggered, and application performance is identical.

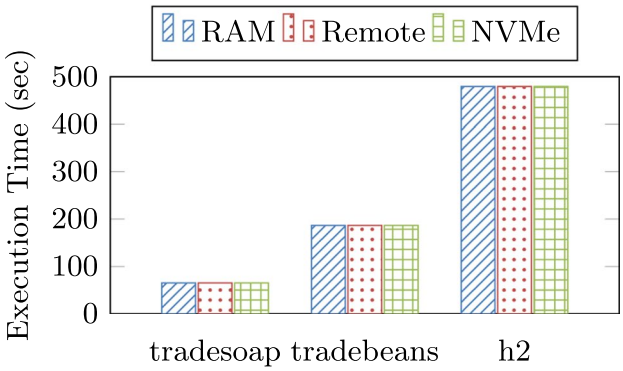
Figure 13 shows results of running selected applications during system overloading. Docker represents running applications in container instance which simulates critical situation, while KVM/QEMU represents running applications in KVM-based VM which simulates practical situation. In simulation of critical situation, container instance has limitation of 512 MiB memory and no limitation on swap space. Thus, most of benchmark applications' data are exchanged to remote memory. In practical situation, there is no limitation for VMs. PM is put into overloading status before running selected applications. Therefore, inactive memory pages of other system processes may be exchanged to remote memory in VM experiments. In addition, applications are killed (out-of-memory) by operating system kernel if remote memory or NVMe SSD based swap is disabled. Thus, there is no performance result of RAM only case.

The results indicate that remote memory has similar performance as local NVMe SSD based swap. Remote memory occasionally has better or worse performance than local NVMe SSD based swap, depending on how an application manages its data in memory. The NVMe SSD based swap has advantages on exchanging large consistent buffers by design, since it can read and write buffers from a clump of blocks. On the other hand, the remote memory has to read or write buffers one by one because InfiniBand sequentially handles the RDMA operations. Moreover, the remote memory has slight advantages on exchanging tiny and scattered buffers, as shown in Fig. 10a, because RAM is truly random access.

In cloud data centers, the remote memory can have both cost and performance advantages over swapping to local disks. Memory sharing reuses spare memory resources located in the remote PM, whilst swapping to local disks requires a dedicated disk space for swapping. In a typical setting of cloud data centers, disks are remotely attached via NAS or SANs, which physically are not a part of the PMs, although they are transparent to the PMs [10, 30]. Swapping to such remotely attached disks cannot bring the same performance as swapping to real local disks attached to PMs. If swapping to SATA-based SSD or HDD, the performance will be much worse than remote memory because these



(a) DaCapo applications in Docker environment.



(b) DaCapo applications in KVM/QEMU environment.

Fig. 12 Comparisons of the execution time performance averaged from 10 runs for DaCapo applications (tradesoap, tradebeans, h2) in Docker and KVM/QEMU environments

types of disk devices are generally much slower than NVMe SSD.

Conclusion

In this paper, we have presented a memory sharing system with which a memory overloaded PM automatically borrows memory resources from a remote PM with spare memory to share out. Our system allows a PM to be over-committed through provisioning more memory resources than its capacity. Unlike traditional methods, such as VM live migration, our system can handle memory overload without interrupting or suspending running applications. In addition, memory sharing is transparent to system processes including applications, VMs, and container instances. We have also designed a unified control algorithm. The borrower PM automatically borrows memory based on a dual-threshold trigger, while the lender reserves sufficient spare memory resources for memory sharing. Experimental studies have been conducted on InfiniBand-networked PMs to demonstrate that the memory sharing system is fully functional as designed. The overall performance of the memory sharing system in the speed and execution time for remote memory access has been shown experimentally to be similar to that for accessing a local NVMe SSD as a swap space. Depending on how an application manages its data in memory, remote memory can occasionally have better or worse performance than local NVMe SSD based swap operations.

This paper has shown the feasibility and full functionality of memory sharing in cloud data centers. Extending the work presented in this paper, our future work will be memory sharing with multiple borrowers and multiple

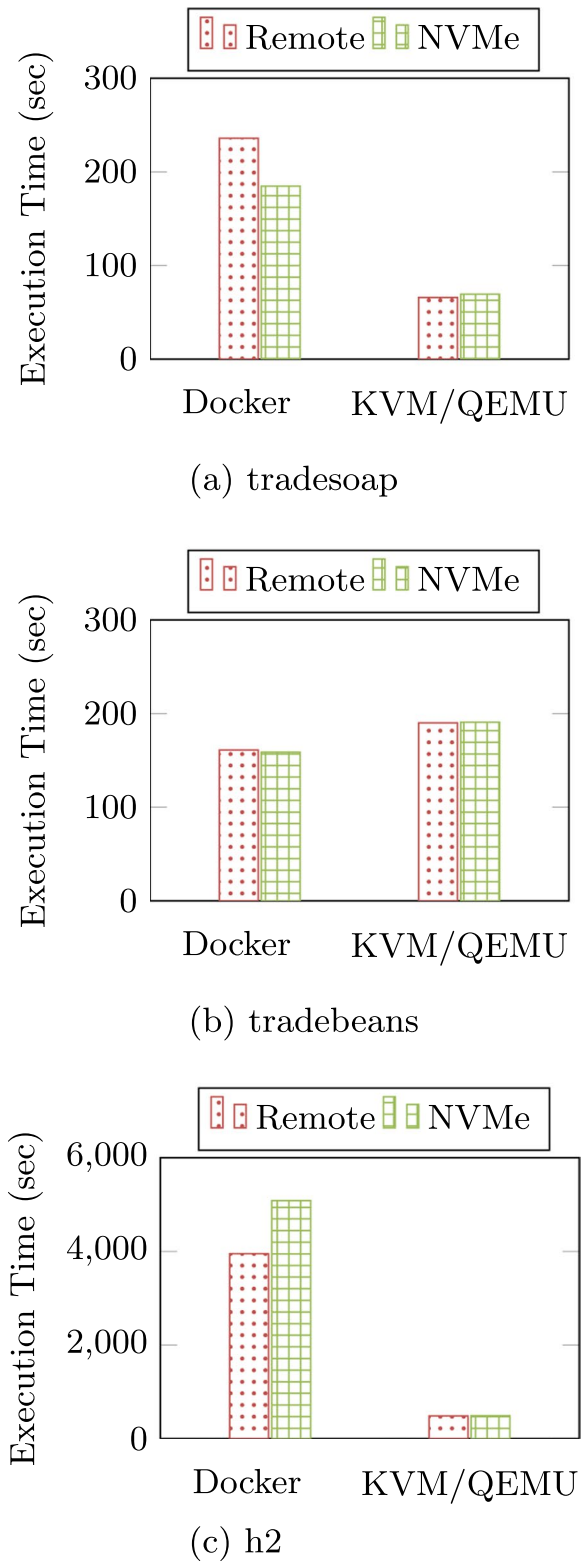


Fig. 13 Execution time performance averaged from 10 runs for DaCapo applications (tradesoap, tradebeans, and h2) in Docker and KVM/QEMU environments

lenders. This will require the planning, scheduling, and optimization of memory resources from a large number of PMs in a data centre. Another future work will be to improve memory sharing for best performance and reliability. This includes the optimization of the virtualized environment, and the reduction of user-kernel space trips.

Abbreviations

CXL	Compute Express Link
HDD	Hard Disk Drive
IaaS	Infrastructure as a Service
KVM	Kernel-based Virtual Machine
NAS	Network-Attached Storage
NQN	NoF Qualified Name
NVMe	Non-Volatile Memory Express
NoF	NVMe over Fabric
OMI	Open Memory Interface
PCIe	Peripheral Component Interconnect Express
PMs	Physical Machines
QoS	Quality of Service
RDMA	Remote Direct Memory Access
SATA	Serial ATA
SLAs	Service-Level Agreements
SSD	Solid-State Drive
SANs	Storage Area Networks
VMs	Virtual Machines

Acknowledgements
Not applicable.

Authors' contributions
Yaozhong Ge: detailed research of this work including formal analysis, methodology and investigation, data processing, and writing for original draft and later revisions. Yu-Chu Tian: project administration, funding acquisition, conceptual project design, project supervision, and writing and editing for original draft and later revisions, and supervision of Y. Ge's research. Zu-Guo Yu: suggestions for methodology and data processing, and writing-reviewing and editing. Weizhe Zhang: suggestions for refined algorithm development and experimental design, and writing-reviewing and editing. The author(s) read and approved the final manuscript.

Authors' information
Y. Ge received the first class Honours degree in information technology from Queensland University of Technology, Brisbane, QLD, Australia, in 2018, where he is currently a Ph.D. candidate with the School of Computer Science. His research interests include big data computing, cloud computing, resource virtualization, machine learning, and distributed systems. Y.-C. Tian received the Ph.D. degree in computer and software engineering from the University of Sydney, Sydney, NSW, Australia, in 2009, and the Ph.D. degree in industrial automation from Zhejiang University, Hangzhou, China, in 1993. He is currently a Professor with the School of Computer Science, Queensland University of Technology, Brisbane, QLD, Australia. His research interests include big data computing, cloud computing, optimization and machine learning, computer networks, smart grid communications and control, networked control systems, and cyber-physical system security. Z.-G. Yu received the Ph.D. degree in mathematics from Fudan University, Shanghai, China, in 1997. He is currently a professor with the School of Mathematics and Computational Science, Xiangtan University, Hunan, China. His research interests include computational mathematics, fractal geometry, complex network analysis, biological and environmental data processing. W. Zhang received the Ph.D. degree in computer science and technology from Harbin Institute of Technology, Harbin, China, in 2006, where he is currently a professor with the School of Computer Science and Technology. He is also with the Cyberspace Security Research Center, Pengcheng Laboratory, Shenzhen, China. His research interests are primarily in cyberspace security, cloud computing, and high-performance computing.

Funding
This work was supported in part by the Australian Research Council (ARC) through the Discovery Project Scheme under Grants DP170103305 and DP220100580 to Y.-C. Tian, in part by the Fundamental Research Funds for the Central Universities under Grant HIT.OCEF.2021007 to W. Zhang, in part by the National Key Research and Development Program of China under Grant 2020YFC0832405 to Z.-G. Yu, and in part by the Science and Technology Innovation Program of Hunan Province of China under Grant 2022WK2009 to Z.-G. Yu.

Availability of data and materials
Not applicable.

Declarations

Ethics approval and consent to participate
Not applicable.

Consent for publication
Not applicable.

Competing interests
The authors declare no competing interests.

Received: 25 July 2022 Accepted: 13 February 2023
Published online: 28 February 2023

References

1. Ahn S, Kim J, Lim E, Kang S (2018) Soft memory box: A virtual shared memory framework for fast deep neural network training in distributed high performance computing. IEEE Access 6:26493–26504
2. Amaral M, Polo J, Carrera D, Gonzalez N, Yang CC, Morari A, D'Amora B, Youssef A, Steinder M (2021) Drmaestro: orchestrating disaggregated resources on virtualized data-centers. J Cloud Comput 10(22):1–20
3. Baset SA, Wang L, Tang C (2012) Towards an understanding of oversubscription in cloud. In: 2nd USENIX Workshop on Hot Topics Manage. Internet Cloud Enterp. Netw. Serv., San Jose, CA, pp 1-6
4. Bell CG, Nassi I (2018) Revisiting scalable coherent shared memory. Computer 51(1):40–49
5. Blackburn SM, Garner R, Hoffmann C, Khang AM, McKinley KS, Bentzur R, Diwan A, Feinberg D, Frampton D, Guyer SZ, Hirzel M, Hosking A, Jump M, Lee H, Moss JEB, Phansalkar A, Stefanović D, VanDrunen T, von Dincklage D, Wiedermann B (2006) The DaCapo benchmarks: Java benchmarking development and analysis. SIGPLAN Not 41(10):169–190
6. Cao W, Liu L (2018) Dynamic and transparent memory sharing for accelerating big data analytics workloads in virtualized cloud. 2018 IEEE Int. Conf. Big Data, Seattle, WA, USA, pp 191–200
7. Cao W, Liu L (2020) Hierarchical orchestration of disaggregated memory. IEEE Trans Comput 69(6):844–855
8. Che J, He Q, Gao Q, Huang D (2008) Performance measuring and comparing of virtual machine monitors. In: 2008 IEEE/IFIP Int. Conf. Embedded Ubiquitous Comput., Shanghai, China, vol 2. pp 381–386
9. Choi HH, Kim K, Kang DJ (2017) Performance evaluation of a remote block device with high-speed cluster interconnects. In: ICCMS'17: Proc. 8th Int. Conf. Comput. Modeling Simul., Canberra, Australia, pp 84–88
10. Choudhary A, Govil MC, Singh G, Awasthi LK, Pilli ES, Kapil D (2017) A critical survey of live virtual machine migration techniques. J Cloud Comput 6(23):1–41
11. Coughlin T (2022) Digital storage and memory. Computer 55(1):20–29
12. Deshpande U, Wang B, Haque S, Hines M, Gopalan K (2010) MemX: Virtualization of cluster-wide memory. 2010 39th Int. Conf. Parallel Process, San Diego, CA, USA, pp 663–672
13. Dragojević A, Narayanan D, Castro M, Hodson O (2014) FaRM: Fast remote memory. In: 11th USENIX Symp. Networked Syst. Design Implementation (NSDI 14), Seattle, WA, USA, pp 401–414
14. Elmeleegy K, Olston C, Reed B (2014) SpongeFiles: Mitigating data skew in Mapreduce using distributed memory. In: SIGMOD'14: Proc. 2014 ACM SIGMOD Int. Conf. Manage. of Data, Snowbird, UT, USA, pp 551–562
15. Everman B, Rajendran N, Li X, Zong Z (2021) Improving the cost efficiency of large-scale cloud systems running hybrid workloads - a case study of Alibaba cluster traces. Sustain Comput Inform Syst 30:100528
16. Gu J, Lee Y, Zhang Y, Chowdhury M, Shin KG (2017) Efficient memory disaggregation with infiniswap. In: 14th USENIX Symp. Networked Syst. Design and Implementation (NSDI 17), Boston, MA, USA, pp 649–667
17. Guo C, Wu H, Deng Z, Soni G, Ye J, Padhye J, Lipshteyn M (2016) RDMA over commodity ethernet at scale. In: SIGCOMM'16: Proc. 2016 ACM SIGCOMM Conf., Florianopolis, Brazil, pp 202–215
18. Guz Z, Li HH, Shayesteh A, Balakrishnan V (2017) NVMe-over-fabrics performance characterization and the path to low-overhead flash disaggregation. In: Proc. 10th ACM Int. Syst. Storage Conf., Haifa, Israel, pp 1–9
19. Healy P, Lynn T, Barrett E, Morrison JP (2016) Single system image: A survey. J Parallel Distrib Comput 90–91:35–51
20. Hines MR, Gopalan K (2007) MemX: Supporting large memory workloads in Xen virtual machines. In: Proc. 2nd Int. Workshop Virtualization Tech. in Distrib. Comput., Reno, NV, USA, pp 1–8
21. Kissel E, Swany M (2016) Photon: Remote memory access middleware for high-performance runtime systems. In: 2016 IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW), Chicago, IL, USA, pp 1736–1743
22. Kocharyan A, Ekane B, Teabe B, Tran GS, Astsatryan H, Hagimont D (2022) A remote memory sharing system for virtualized computing infrastructures. IEEE Trans Cloud Comput. 10(3):1532-1542
23. Koh K, Kim K, Jeon S, Huh J (2019) Disaggregated cloud memory with elastic block management. IEEE Trans Comput 68(1):39–52
24. Li F, Das S, Syamala M, Narasayya VR (2016) Accelerating relational databases by leveraging remote memory and rdma. In: SIGMOD'16: Proc. of the 2016 Int. Conf. on Mgmt. of Data, San Francisco, CA, USA, pp 355–370
25. Liang S, Noronha R, Panda DK (2005) Swapping to remote memory over InfiniBand: An approach using a high performance network block device. IEEE Int. Conf. Cluster Comput, Burlington, MA, USA, pp 1–10
26. Lim K, Turner Y, Santos JR, AuYoung A, Chang J, Ranganathan P, Wenisch TF (2012) System-level implications of disaggregated memory. IEEE Int. Symp. High-Perform. Compt. Archit, New Orleans, LA, USA, pp 1–12
27. Mauch V, Kunze M, Hillenbrand M (2013) High performance cloud computing. Futur Gener Comput Syst 29(6):1408–1416
28. Mehra P, Coughlin T (2022) Taming memory with disaggregation. Computer 55(9):94–98
29. Moltó G, Caballer M, de Alfonso C, (2016) Automatic memory-based vertical elasticity and oversubscription on cloud platforms. Futur Gener Comput Syst 56:1–10
30. Mulahuwaish A, Korbel S, Qolomany B (2022) Improving datacenter utilization through containerized service-based architecture. J Cloud Comput 11(1):1–29
31. Nassi I (2017) Scaling the computer to the problem: Application programming with unlimited memory. Computer 50(8):46–51
32. Newhall T, Lehman-Borer ER, Marks B (2016) Nswap2L: Transparently managing heterogeneous cluster storage resources for fast swapping. In: Proc. 2nd Int. Symp. Memory Syst., New York, NY, USA, pp 50–61
33. Oura H, Midorikawa H, Kitagawa K, Kai M (2017) Design and evaluation of page-swap protocols for a remote memory paging system. In: 2017 IEEE Pacific Rim Conf. Commun. Comput. Signal Process (PACRIM), Victoria, BC, Canada, pp 1–8
34. Ruan Z, Schwarzkopf M, Aguilera MK, Belay A (2020) AIFM: High-performance, application-integrated far memory. In: 14th USENIX Symp. Operating Syst. Design Implementation (OSDI'20), Banff, Alberta, Canada, pp 315–332
35. Schmidl D, Terboven C, Wolf A, Da Mey, Bischof C (2010) How to scale nested openmp applications on the scalemp vsmp architecture. 2010 IEEE Int. Conf. on Cluster Comput, Heraklion, Greece, pp 29–37
36. Srinuan P, Yuan X, Tzeng N (2020) Cooperative memory expansion via OS kernel support for networked computing systems. IEEE Trans Parallel Distrib Syst 31(11):2650–2667
37. Starke WJ, Thompto BW, Stuecheli JA, Moreira JE (2021) Ibm's power10 processor. IEEE Micro 41(2):7–14

38. Waldspurger CA (2003) Memory resource management in VMware ESX server. SIGOPS Oper Syst Rev 36(SI):181–194
39. Wang Z, Wang X, Hou F, Luo Y, Wang Z (2016) Dynamic memory balancing for virtualization. ACM Trans Archit Code Optim 13(1):Art. No. 2, 1-25
40. Williams D, Jamjoom H, Liu YH, Weatherspoon H (2011) Overdriver: Handling memory overload in an oversubscribed cloud. ACM SIGPLAN Not 46(7):205–216
41. Williams D, Jamjoom H, Liu YH, Weatherspoon H (2011b) Overdriver: Handling memory overload in an oversubscribed cloud. SIGPLAN Not 46(7):205–216
42. Wu R, Huang L, Zhou H (2019) Rhkv: An rdma and htm friendly key–value store for data-intensive computing. Futur Gener Comput Syst 92:162–177
43. Xue Y, Zhu Z (2021) Hybrid flow table installation: Optimizing remote placements of flow tables on servers to enhance PDP switches for in-network computing. IEEE Trans Netw Serv Manage 18(1):429–440
44. Younge AJ, Reidy C, Henschel R, Fox GC (2016) Evaluation of smp shared memory machines for use with in-memory and openmp big data applications. In: 2016 IEEE Int. Parallel and Distributed Processing Symp. Workshops (IPDPSW), Chicago, IL, USA, pp 1597–1606
45. Zhang F, Liu G, Fu X, Yahyapour R (2018) A survey on virtual machine migration: Challenges, techniques, and open issues. IEEE Commun Surv Tutor 20(2):1206–1243
46. Zhang W, Xie H, Hsu C (2017) Automatic memory control of multiple virtual machines on a consolidated server. IEEE Trans Cloud Comput 5(1):2–14
47. Zhao W, Wang Z, Luo Y (2009) Dynamic memory balancing for virtual machines. SIGOPS Oper Syst Rev 43(3):37–47

Publisher's Note