A Remote Memory Sharing System for Virtualized Computing Infrastructures

Aram Kocharyan, Brice Ekane, Boris Teabe, Giang Son Tran, Hrachya Astsatryan, and Daniel Hagimont

Abstract—Resource management is a critical issue in today's virtualized computing infrastructures. Consolidation is the main technique used to optimize such infrastructure. Regarding memory management, it allows gathering overloaded and underloaded VM on the same server so that memory can be mutualized. However, because of infrastructures constraints and complexity of managing multiple resources, consolidation can hardly optimize memory management. In this article, we propose to rely on a remote memory sharing for mutualizing memory. We implemented a system which monitors the working set of virtual machines, reclaims unused memory and makes it available (as a remote swap device) for virtual machines which need memory. Our evaluations with HPC and Big Data benchmarks demonstrate the effectiveness of this approach. We show that remote memory can improve the performance of a standard Spark benchmark by up the 17 percent with an average performance degradation of 1.5 percent (for the providing application).

Index Terms—Virtualization, remote memory, sharing

1 Introduction

An increasing number of applications require huge amounts of computational and data resources provided by large scale hardware infrastructures, i.e., clusters of servers. As such infrastructures can be mutualized, this contributes to the development of data centers following the cloud computing model. Such data centers may target the public market (public clouds) or be operated by companies for their internal use (private clouds).

A majority of these clusters rely on virtualization for resource management simplification [1]. Virtualization allows hosting several virtual machines (VMs) / operating systems on a single physical machine. Each VM is allocated a given amount of resources (CPU, memory, networking, storage) and it represents the unit of allocation in the infrastructure. Thanks to VM migration, a consolidation policy can potentially be implemented [2], which consists of packing VMs on as less physical servers as possible.

In this context, two main types of resource intensive applications are generally exploited: High Performance Computing (HPC) and Big Data (BD). HPC applications are mainly CPU bound [3] while BD applications are mainly IO and memory bound [4].

This paper aims to improve the memory management in such environments. The generally adopted approach is to monitor the working set of each VM and to reclaim weakly used memory (cold pages) without degrading the VM performance. Then, the reclaimed memory can be given to VMs with high memory requirements [5]. However, this can only be done on a per server basis as reclaimed memory on one server can only be given to VMs running on that server. Therefore, we have to trust the placement and consolidation systems for gathering on the same server memory providing VMs and memory consuming VMs.

However, this approach is difficult to implement for two main reasons:

1) Consolidation limitations. Consolidation is known to be a NP hard problem [6], especially since it has to simultaneously take into account multiple resource types whose availability is continuously varying. Therefore, it is a challenge to colocate VMs so that memory can be mutualized.
2) Infrastructure concerns. VMs' placement may be constrained by rules linked with the hardware type or with administration policies (e.g., different sub-clusters for HPC or BD applications), thus limiting the use of VM migration and dynamic consolidation.

Therefore, requiring VM colocation for memory mutualization appears to be a substantial limitation. The principle followed by the suggested contribution is to make the reclaimed memory accessible remotely.

In this paper, *we present the implementation of a system which allows to (1) dynamically monitor the working set of each VM, (2) to aggregate this memory into a distributed memory reservoir, and (3) to make it available to requiring VMs. This memory can be used directly by the VM if available locally. It can be used as a fast remote swap device when available remotely*. Our

- *Aram Kocharyan and Hrachya Astsatryan are with the Institute for Informatics and Automation Problems, National Academy of Sciences, Yerevan 0019, Armenia. E-mail: ar.kocharyan@gmail.com, hrach@sci.am.*
- *Brice Ekane is with the University Grenoble Alpes, 38400 Saint-Martin-d' Hères, France. E-mail: brice.ekane@gmail.com.*
- *Boris Teabe and Daniel Hagimont are with the University of Toulouse, 31000 Toulouse, France. E-mail: {boris.teabedjomgwe, daniel.hagimont}@inp-toulouse.fr.*
- *Giang Son Tran is with the ICTLab, University of Science and Technology of Hanoi, Hanoi 100000, Vietnam. E-mail: tran-giang.son@usth.edu.vn.*

Manuscript received 10 Dec. 2019; revised 30 June 2020; accepted 5 July 2020. Date of publication 21 Aug. 2020; date of current version 6 Sept. 2022. (Corresponding author: Boris Teabe.) Recommended for acceptance by V. Piuri. Digital Object Identifier no. 10.1109/TCC.2020.3018089

evaluations with HPC and BD benchmarks demonstrate the effectiveness of this approach. We show that a remote memory reservoir provided by a HPC cluster through an Infiniband network can improve the performance of a standard Spark BD application by up to 17 percent with an average performance degradation of 1.5 percent (for the providing application).

The rest of the article is structured as follows. Section 2 presents the motivations and main design choices. Section 3 details the designed and implemented system. Section 4 presents the performance evaluation which demonstrates the effectiveness of the approach. After a review of related works in Section 5, we conclude the article in Section 6.

2 Motivation and Design Choice

2.1 Motivation

An increasing number of applications require huge amounts of resources, especially HPC (CPU bound) and BD (IO and memory bound) applications. Consequently, companies are setting up private cloud infrastructures (datacenters) where the resources required by such applications can be mutualized. For ease of administration, most of these infrastructures are virtualized. Therefore, VMs are the unit of resource allocation and placement in the datacenter.

Resource allocation to VMs can be static or dynamic:

1) static allocation: resources are allocated to each VM at creation time and never reclaimed during the lifetime of the VM. This approach is widely used due to its ease of use. However, it leads to a waste of resources, since users estimate and allocate the maximum amount of resources needed for their applications to prevent performance degradations during the peak workloads. Thus, when VMs' workloads are lower, the allocated resources stay unutilized.
2) dynamic allocation: even if a VM is configured with a given amount of resources (a maximum), resources are effectively allocated on demand. This allows to mutualize the resources within one node and increase resource utilization, since resources which are not used by a VM can be reclaimed and reused for another VM.

Finally, thanks to VM migration, the placement of VMs in the datacenter can be modified dynamically, following a consolidation strategy. Consolidation consists in packing VMs on as less physical servers as possible, and aims at optimizing resource management. Such consolidation can be implemented either with static or dynamic resource allocation. Most of the works regarding consolidation were either based on static allocation or dynamic allocation, but considering for most of them the CPU resource only. Addressing the consolidation problem with dynamic allocation and multiple fluctuating resources is a much tricky issue.

In this paper, we are interested in improving memory management in such environments. We assume a dynamic allocation approach where we monitor the working set of each VM and reclaim weakly used memory (cold pages) without degrading the VM performance. Then, the reclaimed memory can be given to VMs with high memory requirements. However, this can only be done on a per server basis as reclaimed memory on one server can only be given to VMs running on that server. Therefore, we have to trust the placement and consolidation systems for gathering on the same server memory providing VMs and memory consuming VMs.

However, this approach is difficult to implement for three main reasons:

1) Consolidation limitations. As we have seen previously, consolidation with dynamic allocation and multiple resources is tricky and it is known to be a NP hard problem [6], especially since it has to simultaneously take into account multiple resource types whose availability is continuously varying. Therefore, many providers renounce using consolidation and rely on simple predictable policies, thus making it difficult to colocate VMs so that memory can be mutualized.
2) Infrastructure concerns. VMs' placement may be constrained by hardware type concerns, e.g., the availability of a specific device like a GPU. Or sometimes, the hardware may just not enable VM migration, e.g., SR-IOV network devices [7] significantly reduce the overhead of virtualization, but forbid migration in major hypervisors.
3) Administration concerns. Different parts of the infrastructures may be dedicated to different types of application, e.g., different sub-clusters for HPC or BD applications. Indeed, the infrastructures built to run HPC and BD applications are quite different from one another. In HPC clusters, there is generally a centralized file system which is shared among all the (diskless compute) nodes via NFS (or a similar technology). In Big Data clusters, every node has its own local storage device because every node accesses its local storage intensively. Moreover, the schedulers used for distributing jobs in these clusters are different due to the difference in the type of job they distribute. Consequently, migrations between these sub-clusters can hardly be operated.

Therefore, relying on consolidation to enforce VM colocation for memory mutualization appears to be an hazardous strategy. The principle followed by our contribution is to make the reclaimed memory accessible remotely.

2.2 Design Choice

In this paper, we present a memory mutualization system which relies on dynamic memory allocation. The working set of each VM is monitored and weakly used memory can be reclaimed. This reclaimed memory can be used to provision VMs which lack memory on the same server. It can also be used to provision a memory reservoir which behaves as a remotely accessible fast storage. Modern networking technologies (such as Infiniband) provide low latency and high bandwidth communication allowing to use this memory as a fast swap device. Therefore, the free memory reservoir is used as an extension of the local memory of VMs.

We observed that VMs in HPC clusters are mainly CPU bound and their memory consumption is quite stable, allowing memory to be reclaimed to provision the memory reservoir. Most applications in BD clusters are memory and

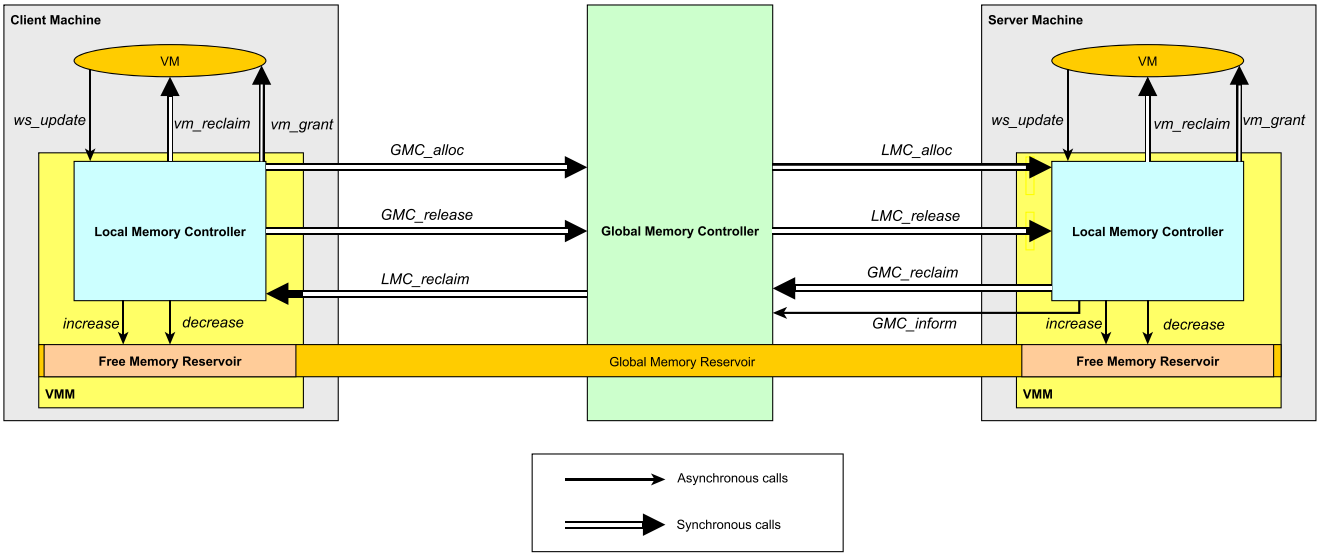


Fig. 1. Architecture of the distributed memory sharing system.

IO bound and can significantly benefit from extra memory from the memory reservoir.

The next sections present the design of our system and its evaluation.

3 Contribution

In the presented distributed memory sharing system, memory may be mutualized locally (to a node) or globally (between remote machines). In the case of local mutualization, unused memory from local VMs can be used to help overloaded local VMs. In the case of global mutualization, unused memory from VMs on one node can be used (as a fast swap) to mitigate memory shortage of remote VMs.

Physical machines act either as a Client (memory consumer) or a Server (memory provider). A client machine can benefit from remote memory from server machines. A machine which does not use all of its memory becomes a server. A machine which requires more memory (than its capacity) becomes a client. However, every VM is guaranteed to have at least its initially allocated memory in case of memory shortage. Thus VMs and client machines can get their memory back in such cases.

The suggested system is composed of two parts: dynamic memory allocation within one node (local memory mutualization) and remote memory allocation from server machines (global memory mutualization).

3.1 Design

3.1.1 Overall System Architecture

The design of our system relies on two main entities:

- A Local Memory Controller (*LMC*) is in charge of memory management within a single node. Every node (client or server) is running an *LMC*. The *LMC* manages a *Free Memory Reservoir*. This memory may be used for local or global mutualization.
- A Global Memory Controller (*GMC*) manages the coordination between machines (clients and servers). It is connected with all the *LMCs*. It implements a *Global Memory Reservoir* by federating the distributed *free memory reservoirs*. It is responsible for remote memory distribution among clients.

Fig. 1 illustrates the architecture of the distributed memory sharing system. Each single machine executes an instance of *LMC*, which is responsible for local memory management within the node. The virtualization system includes a memory monitoring system which periodically evaluates the working set size of each VM. The estimations are transmitted to the *LMC* which is allowed to reclaim or grant memory from/to VMs depending on their memory needs and current allocations. The memory reclaimed by the *LMC* provisions the *Free Memory Reservoir* (local to the machine). This memory can be allocated to local VMs. The information on the size of the *Free Memory Reservoir* is sent (when modified) to *GMC* which implements the *Global Memory Reservoir*. This memory can be allocated to client machines (their *LMC*) and then be used as a remote swap device by overloaded VMs.

LMCs in all machines keep communication with the *GMC* for requesting and releasing remote memory. This communication can rely on different types of networks. We experimented both with Gb Ethernet and Infiniband. Infiniband brings the advantage of enabling Remote Direct Memory Access (RDMA). With Infiniband, the communication framework between *LMC* and *GMC* implements the concepts of RPC over RDMA [8], [9]. Communications between VMs and remote swap devices rely on low-level RDMA primitives which directly access remote memory.

3.1.2 Monitoring Service

A proper working set estimation facility for the memory management system should have a very low performance impact on VMs. It should also be accurate regarding the estimation of the working set size (WSS). For implementing such a facility, the main issues is to enforce accuracy and low overhead in two states, when a VM does not use all its memory (S_{less}) and when it lacks memory (S_{more}).

In our system, we rely on badis [10], a WSS monitoring system implemented in our research group. Badis combines a

statistical working set estimation method based on *page invalidation* [11] and a *buffer cache* monitoring based method [12].

Badis accurately estimates WSS in both the S_{less} case (with the *page invalidation* method) and the S_{more} case (with the *buffer cache* method). Thus, these solutions complete one another.

3.1.3 Memory Management Controllers

The memory management protocol defines the interactions between controllers (*LMC* and *GMC*) and between *LMCs* and the virtualization system. The protocol is composed of two parts:

1) the Local Memory Management Protocol defines the exchanged messages between the virtualization system and the *LMC*. It allows the *LMC* to obtain the WSS from the virtualization system and to grant or reclaim memory to/from a VM.
2) the Global Memory Management Protocol defines the exchanged messages between *GMC* and *LMC*. It allows a client *LMC* to request (to the *GMC*) the allocation or the release of extra memory, or it allows a server *LMC* to reclaim its memory back. A *LMC* can also inform the *GMC* about the size of the local reservoir. Symetrically, the *GMC* can request (to a server *LMC*) a memory allocation or release, or it can reclaim some memory back.

The defined protocol is described in Fig. 1, with one arrow per message type.

Local Memory Controller Behavior. The *LMC* periodically gathers the WSS from all the VMs running on the machine. The *ws_update* message is used to receive the WSS information of all VMs. Based on this information, the *LMC* decides wether to *grant* extra memory to VMs or *reclaim* memory from VMs. The implementation of *vm_grant* and *vm_reclaim* are based on memory ballooning [11], [13] (using a balloon driver installed in all the VMs). These operations naturally force a change of the VM sizes and a change of the size of the *Free Memory Reservoir*: a *vm_reclaim* (respectively *vm_grant*) on a VM increases (respectively decreases) the size of the *Free Memory Reservoir*. Later, the memory gathered in the *Free Memory Reservoir* may be provided as a remote swap device.

Global Memory Controller Behavior. The *GMC* periodically receives updates on the current size of all the *Free Memory Reservoir*. The *GMC_inform* message is used to notify every modification of the size of a *Free Memory Reservoir*. When a machine needs extra memory (a remote swap device) for one of its VMs, the *LMC* of the client machine sends a *GMC_alloc* message to the *GMC* asking to check if it may provide the needed amount of memory. *GMC* is aware of the current state of the *Free Memory Reservoirs* in all machines, thus it can decide where it can allocate a remote swap device for a client machine. Then, it sends an *LMC_alloc* message to the chosen server machine (which can provide memory to the client machine). The *LMC* (in the server machine) allocates the memory and prepares it as a swap device and returns all the necessary information to the *GMC*. After a successful allocation on the server side, the *GMC* returns to the *LMC* on the client machine all necessary information allowing to mount and use the remote swap device which is physically located on a server machine. If an *LMC* of a client machine does not need its extra memory anymore, it can release it with a *GMC_release* message, which propagates to the *LMC* on the server machine with a *LMC_release* message. In the case where a server machine needs its memory back, its *LMC* sends a *GMC_reclaim* message to the *GMC*. Then, the *GMC* notifies the client machine that the remote swap device has to be unmounted (*LMC_reclaim* message) and then returns to the server machine that the memory is free. To avoid high latencies that we would have if you were moving the data from the removed swap device, we asynchronously store the data of a remote swap device on a local disk (on the client machine). Thus, the local disk can replace a remote swap device (rapidly and temporarily as the data will potentially be cached in another remote swap device) in such cases.

3.1.4 Remote Swapping Service

Swapping is a key functionalities provided by Linux OS. It uses swap space to increase the amount of virtual memory available to a machine. When the OS faces memory shortage, some pages have to be swapped out to local disk. The OS invokes the *kswapd* kernel module which is responsible for paging. It sends pages to the swap device having the highest priority in the list of devices, which performs I/O operations that are specific for the given type of device. The remote swapping service implements a new type of device which operates with the same logic and gets the highest priority among swap devices. Applications are not aware of this process. Remote swapping is used to provide the unused memory collected in the *memory reservoir* to remote machines.

3.2 Implementation

3.2.1 Implementation Environment

Xen [13] is a popular open-source virtualization system which is widely espoused by several cloud providers such as Amazon EC2. Its implementation follows the para-virtualization [14] model. In the latter, VMs' OS are modified to be aware of the fact that they are virtualized, which reduces virtualization overhead. In this model, a small kernel called the *hypervisor* runs directly on top of the hardware, so taking the traditional place of the OS. Thus, it has all privileges and rights to access the entire hardware and provides the way to run several OS called Virtual Machines (VMs) concurrently. The host OS (seen as a special VM) is called the *Virtual Machine Monitor* (VMM). It has much more privileges than other VMs since it is responsible for running Xen management toolstack.

3.2.2 Monitoring Service

As mentioned before for working set monitoring, we rely on Badis [10], a WSS monitoring system implemented in our research group. Fig. 2 illustrates the architecture of Badis.

The *Page invalidation* method is inspired by a technique used in VMware [11]. It is based on the following logic: it periodically invalidates a random set of pages and counts the number of accessed pages by the end of each period. Based on these numbers, it statistically calculates the WSS.

The *buffer cache* method is inspired by a technique provided by Geiger [12]. It monitors the evictions and subsequent

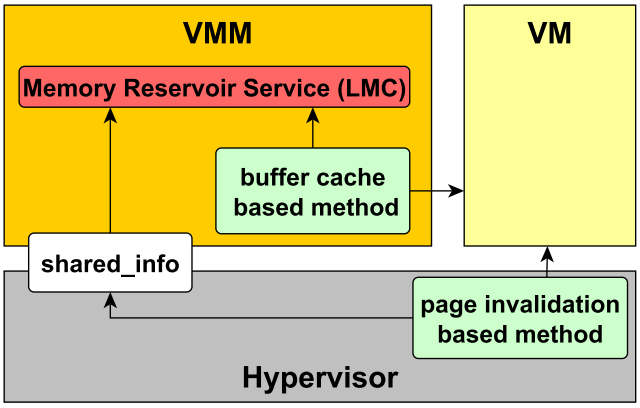


Fig. 2. The architecture of Badis.

reloads from the guest OS buffer cache to/from the swap device. It relies on the ghost buffer [15] method to estimate the WSS.

The implementation of the *Page invalidation* method is located in the kernel of the hypervisor while *buffer cache* is implemented in the kernel of the VMM [13]. Both of them keep communication with the *LMC* to transmit the WSS values. The data transfer between the *buffer cache* module and the *LMC* is facilitated because they are both located in the kernel of the VMM. The data transfer between the *Page invalidation* module and the *LMC* required changes in the native Xen share_info data structure, which allows to share small amounts of data needed to boot a VM.

3.2.3 Memory Reservoir Service

This service is composed of two parts: the *LMC* and the *GMC*. The *LMC* is implemented as a loadable kernel module installed in the VMM in order to allow interactions with VMs for monitoring and management. The *GMC* can be run in any process on any machine in the cluster.

The *LMC* implements dynamic memory allocation based on the memory ballooning technique which is a simple driver located in the VM (guest OS). The driver communicates with the VMM and may receive two types of command: *inflate* and *deflate*. In case of *inflate* command, the driver allocates memory in the VM and gives its control to the VMM (so that the *LMC* can use it). In case of *deflate* command, the VMM releases its control over pages, and the driver can deallocate these pages (therefore making these pages available in the VM). This technique is used to implement dynamic memory allocation (to VMs) in our system.

The *GMC* is managing its communication via standard network interfaces such as Ethernet and Infiniband. The communication via Ethernet is implemented by using low level netpoll APIs, which allow to send UDP packets from the Linux OS Kernel. In case of communication over RDMA (Infiniband), the standard RDMA *SEND/RECV* functions from the IBverbs library are used.

3.2.4 Remote Swapping Service

In the guest OS of a VM, *kswapd* can use swap devices. Such a swap device is implemented in the VMM on the local host. Remote swapping is implemented with two modules, on the client side (memory consumer) and on the server side (memory provider):

1) The client side module is located in the VMM of the client host and provides a means for *kswapd* to swap pages to the swap device.
2) The server side module is located in the VMM of the server host and is responsible for allowing reads and writes to the memory reservoir from remote locations.

Fig. 3 describes the architecture of the remote swapping service. These two modules can be interconnected via the Infiniband network which makes possible remote swapping without interaction with the remote CPU. For its implementation on Infiniband, remote swapping is based on Infiniswap [16] where *RDMA READ* and *RDMA WRITE* requests allow direct addressing of remote pages.

3.3 Memory Management Policy

The *LMC* adjusts the memory size of VMs according to the working set values. If a VM is over-provisioned, then unused memory (according to the working set) is reclaimed and sent to the *Free Memory Reservoir*. In case of memory shortage of a VM, the memory needs can possibly be satisfied from the *Glocal Memory Reservoir*, locally or remotely. The system guarantees that at least the VM's initially allocated memory size is provisioned if its working set grows up to that size, and some extra memory can be allocated if the *Global Memory Reservoir* is not empty.

There are many configurable parameters in our system, which give the opportunity to tune the system according to

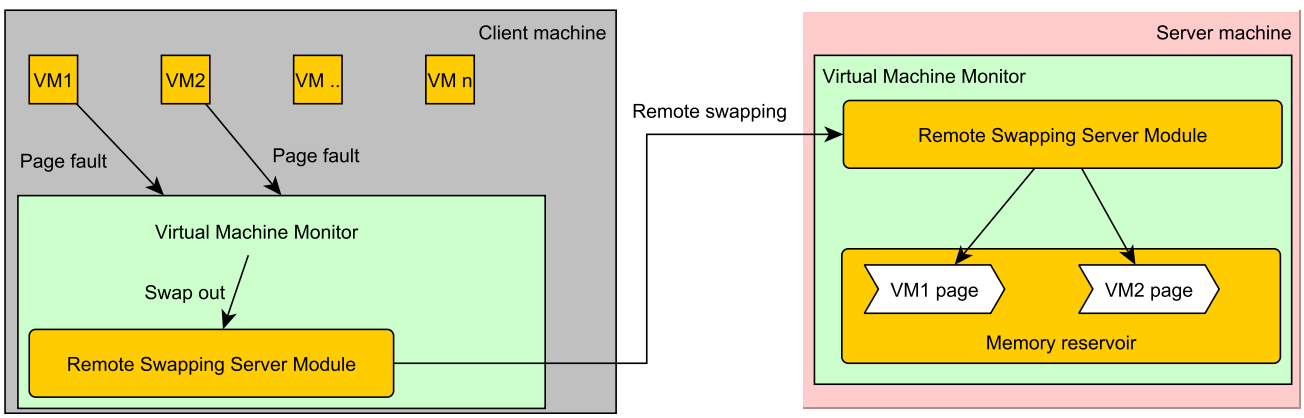


Fig. 3. The architecture of remote swapping.

the needs of an administrator. Some administrators would give priority to local VMs (satisfy local needs first), others would give higher priority to a given type of VM (potentially remote).

In our case, we tried to satisfy local needs first and a *Free Memory Reservoir* is always used as a remote swap device when all the VMs on that host are satisfied. In our policy, we also added a configurable limit (per VM) on the amount of memory that can be provided as a remote swap device.

3.4 Memory Allocation for Application

Our distributed memory sharing system allows VMs to obtain extra memory. This will change the size of the VM dynamically, i.e., the amount of memory granted to the VM.

We can consider two situations regarding the use of this extra memory by applications:

- Applications with dynamic memory allocation. These are applications which dynamically allocate and release memory. Such application will naturally benefit from the extra memory granted to the VM.
- Applications with static memory allocation. These are applications which allocate as much memory as possible at startup and implement their own memory management internally. The Java virtual machine is an example of such application. Therefore, such applications will not benefit from extra memory granted to the VM.

Regarding the second class of application, we use our system in the following way. First, we monitor the behavior of VM images to determine the type of VM which needs extra memory (e.g., Big Data VMs). These VMs are configured with an increased memory size, so that they will use extra memory. We show in the evaluation that a Big Data VM improves its performance with up to 40 percent of extra memory. For such VMs, the extra memory must be available if the VM uses it, else the VM would swap to disk and it would degrade performance. Second, the quantity of extra memory which can be allocated is based on statistics. We monitor the use of memory in the datacenter and therefore have an estimation of the extra memory which can be allocation without damage.

The question of which VM should receive extra memory (static application VMs, dynamic application VMs, which VM) is a question of policy.

4 Evaluation

This section presents the details and results of our evaluation. The provided numbers are the average of ten executions. We do not provide standard deviation as it was not significant.

4.1 Methodology

This subsection presents the benchmarks we used and evaluation methodology we relied on. In our evaluations, we consider the client side where remote memory may be used, and the server side from where remote memory is provided. On the client side, we evaluate the performance benefit with memory intensive applications when additional memory is provided. On the server side, we evaluate the performance degradation since that memory is used remotely. On the server side, applications are not memory intensive applications, but rather CPU intensive applications which don't use all their allocated memory.

On the client side, we chose the following types of application for the evaluation of our system:

1) *Microbenchmark* - a simple application that we implemented, whose memory behavior is known in advance, thus allowing to precisely evaluate the contribution.
2) *Memory Intensive benchmarks* - standard benchmarks which are known to stress memory. In this regards, we chose the following benchmarks: Data Caching from CloudSuite [17], Elasticsearch nightly benchmarks [18] and BigBench [19].
3) *Big Data benchmarks* - modern memory intensive benchmarks which are used in the domain of Big Data computing. The following benchmarks have been chosen: Spark SQL [20], TestDFSIO [21] and SparkPi from Spark bench [22].

On the server side, we evaluate the performance degradation using the High Performance Linpack [23] benchmark which is used to evaluate performance on Top 500 supercomputers.

The results we present in this section contains the estimation of the working set size, the memory reservation and the remote memory usage. That is, when we ran all benchmarks (client side and server side), our system first estimate the working set size, then reserve memory and finally configure this for remote usage. All the results presented in this section integrate all these steps.

4.2 Experimental Environment

Hardware. We evaluated our contribution on Dell PowerEdge R610 servers with the following configuration: Intel(R) Xeon E5-2630LV4 CPU, 64 GB of memory, 4 x 512 GB of SSD storage. The servers were used with the following scenario: one machine hosts the *GMC*, one machine acts as a Server and one machine acts as a Client.

Virtualization. The virtualization environment we used is Xen 4.2. All the VMs are running an Ubuntu Server 12.04 with Linux kernel 3.6. In all our experiments, the VM configuration that we used is a *medium* size VM, as defined by Amazon web Services: 2 VCPU and 4 GB of memory.

Network. Our default implementation relies on the Infiniband network. Our evaluations were performed with Mellanox ConnectX-3 cards. However, to evaluate the impact of Infiniband, we also performed an evaluation with Ethernet networks.

Software. For benchmarks which rely on Java, the Java Virtual Machine (JVM) configuration (maximum memory allocation) has been changed to make the JVM aware of extra memory available in the remote memory reservoir. Some of the testing applications are based on Apache Spark. In this regard, we evaluated our contribution both with standard apache spark and with an improved version (presented in Xiaoyi [24]) of Spark with RDMA (Infiniband) implemented as a plug-in in Spark which overrides the shuffle methods (this optimized version was implemented at Ohio University, therefore we call it *Ohio Spark*). Comparing with this RDMA optimized version of Spark is a means to know where the benefit comes from.

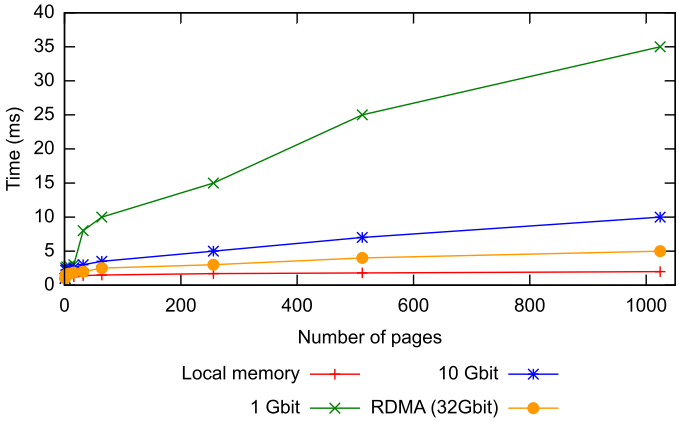


Fig. 4. Latency while accessing pages in different locations.

4.3 Evaluation Results

4.3.1 Micro-Benchmarks

Micro-benchmark loops and accesses (reads or writes) on every cell of an array that has a fixed size assigned at creation time. Every cell in the array has the size of a page (4 Kb). The performance metric of this benchmark is the latency.

Fig. 4 shows the latency of Micro-benchmark while accessing pages locally or remotely using several network technologies (1 Gbit, 10 Gbit, Infiniband with RDMA 32 Gbit). The overhead caused by remote memory location with a small amount of data is not significant. Logically, when the data size increases, the latency increases and the overhead caused by network communications becomes significant. At the level of 1,024 pages, using the 1 and 10 Gb networks makes the application respectively 17 and 5 times slower compared to local memory. We can also observe that the Infiniband network with the RDMA technology can minimize the network overhead down to an acceptable level. During our experiments, we observed that RDMA is only 50 percent slower than local memory.

4.3.2 Memory Intensive Benchmarks

We performed experiments with memory intensive benchmarks which are known to stress memory. The following benchmarks were used:

Data Caching [17] uses the Memcached data caching server to simulate the behavior of a Twitter caching server using a Twitter dataset.

Elasticsearch Nightly Benchmarks [18] is a benchmark suite. We only performed evaluations for the NYC taxi benchmark. The NYC taxi data set contains the rides that have been performed in yellow taxis in New York in 2015. This benchmark evaluates the performance of Elasticsearch for structured data.

BigBench [19] includes more than 30 queries. We chose query 23 because it has the longest execution time.

For each benchmark, we designed 4 types of workloads to investigate the correlation between performance improvement and swap activity. Each workload determines the amount of memory used by the benchmark.

1) *zero* is a workload where the memory allocated to the VM is enough to execute the application.

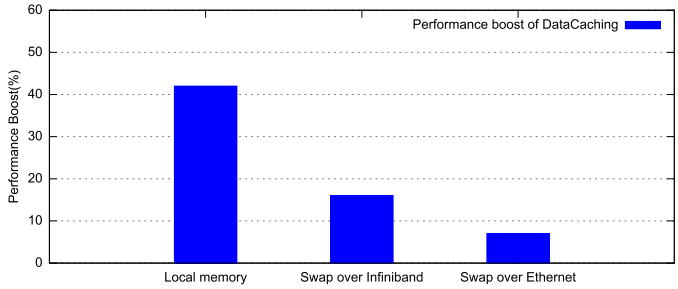


Fig. 5. Data Caching benchmark performance boost with memory extension (local, Infiniband, Ethernet).

2) *light* is a workload where the used memory exceeds the memory allocated to the VM so that up to 10 percent of its memory goes to swap.
3) *medium* is a workload where up to 20 percent of its memory goes to swap.
4) *heavy* is a workload where up to 30 percent of its memory goes to swap.

Fig. 5 shows the performance improvement obtained with the Data Caching benchmarks with a heavy workload. The baseline is the execution without memory extension, so that the required swap is managed on disk. We measured the improvement when the VM is given its (required) memory extension (1) with local memory (2) with remote memory (swap) through Infiniband and (3) with remote memory (swap) through Ethernet (10 Gb). A local memory extension would bring an improvement of 40 percent, which is the ideal case and corresponds to local memory mutualization. We observed that with remote memory, we still have a significant improvement with Infiniband (17 percent) and with Ethernet (8 percent). It demonstrates the interest to mutualize memory even remotely.

Fig. 6 shows the performance improvement for our 3 selected memory intensive benchmarks with the different workload sizes, when being provided memory extension over Infiniband. Naturally, the improvement is proportional to the memory extension required by the workload. We observe that the improvements are significant for all benchmarks.

One important aspect is that the memory reservoir is provisioned by memory reclaimed on the server side. The reclamation and the remote use of such memory may have a

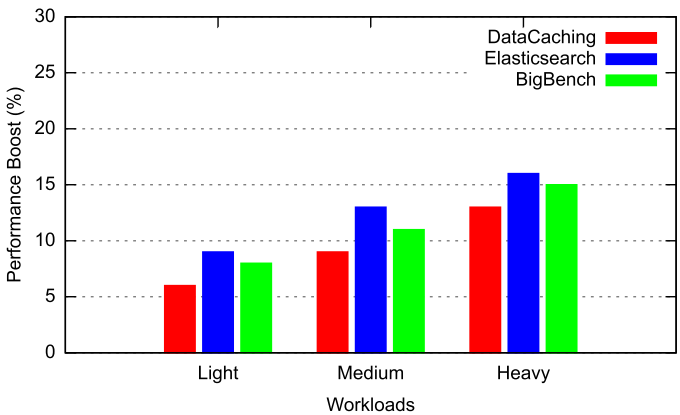


Fig. 6. Performance boost of memory intensive benchmarks with memory extension on Infiniband.

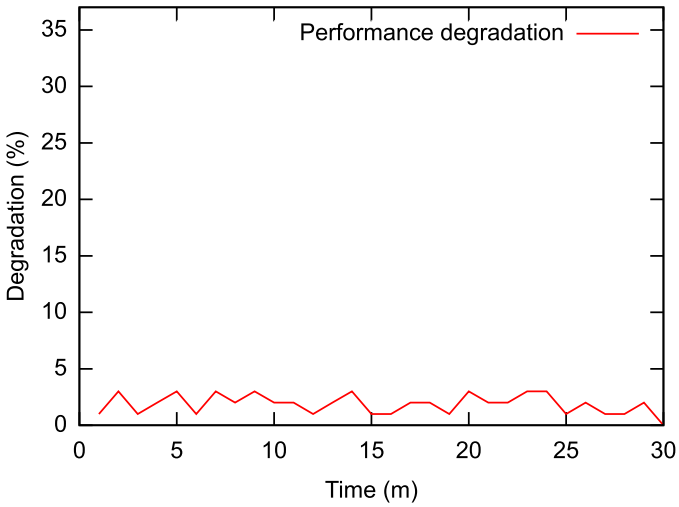


Fig. 7. Performance degradation for server side VMs executing the HPL benchmark.

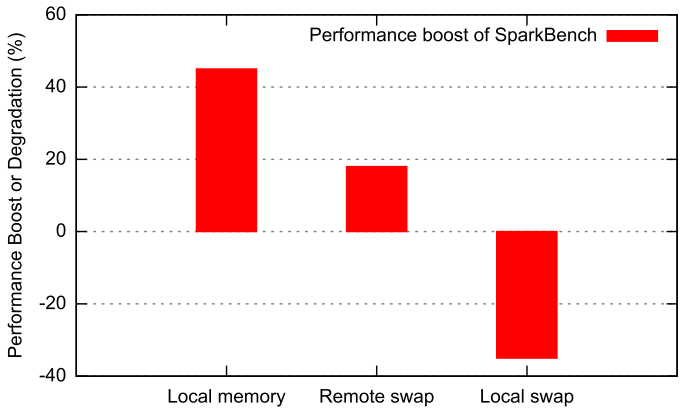


Fig. 8. Performance evaluation of SparkBench when extending memory with different types of memory.

negative impact on applications running on the server side. Remind that such applications are not memory intensive applications but rather CPU intensive applications which do not fully use their memory. We have used the HPL [23] benchmark to evaluate the performance degradation on the server side. On Fig. 7, we can see that the performance degradation is in the 1-3 percent range which is fairly low.

4.3.3 Spark Benchmarks

We also experimented with modern memory intensive benchmarks which are used in the domain of Big Data computing. All the benchmarks run on Spark. We relied on the following benchmarks:

Spark SQL [20] is a Spark extension (as module) that enables to support relational processing to benefit from the advantages of relational processing and Spark analytical libraries. As a workload, we have selected JoinPerformance [25] which compares the performance of joining different table sizes and shapes with different join types.

TestDFSIO [21] is a benchmark that attempts to measure the capacity of HDFS for reading and writing bulk data.

SparkBench [22] is a benchmark suit for benchmarking and simulating Spark jobs. In this paper, the evaluation has been made with PageRank benchmark due to its memory intensive behavior. It is based on the algorithm which was used by Google as an initial algorithm for ranking web pages according their significance and their back-links.

In the case of Spark applications, Spark adapts its memory consumption according to the memory available in the VM. Therefore, for each of these benchmarks, we evaluated the impact of providing a memory extension to the VM. For all experiments, memory was extended by 30 percent of the original memory size of the VM, assuming that this memory is statistically available in the cluster.

Fig. 8 presents the performance impact on SparkBench when extending the memory allocated to the VM with a different type of memory: (1) with local memory (2) with remote memory (swap) through Infiniband (default Spark) and (3) with a local disk (swap). Again here, local memory is the ideal case, but we can see that remote memory on Infiniband also brings a significant benefit (about 19 percent). It also confirms that local disk swap is a bad idea, but it should not happen if the memory extension is allocated based on statistics about used memory in the cluster (Section 3.4).

Fig. 9 shows the performance improvement for our 3 selected big data benchmarks when being provided memory extension over Infiniband. The evaluation was performed both with the default implementation of Spark (Fig. 9 left) and with the OHIO Spark version (Fig. 9 right). We observe that remote memory extension allows a performance improvement from 10 to 17 percent with default Spark, and from 7 to 10 percent with OHIO Spark. It is worth to mention that even with an optimized version of Spark which fully takes advantage of RDMA, remote memory extension still brings a significant benefit.

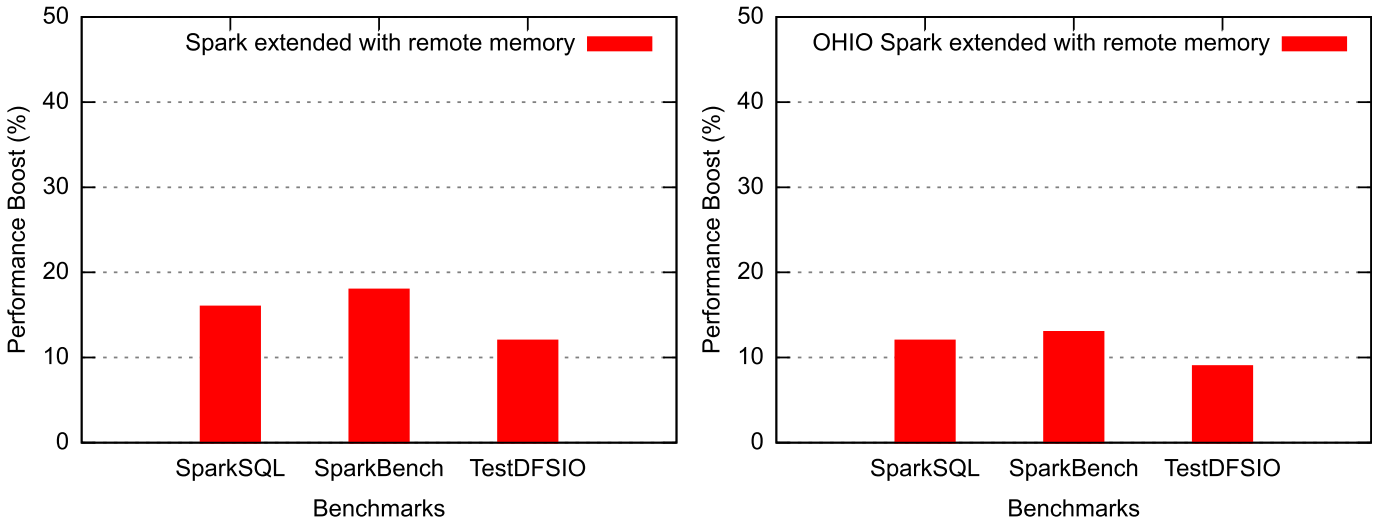
It is well known that Spark is considered to be much efficient since it does its computations in memory. In this regard, it is obvious that the most valuable resource for Spark applications is memory. Thus, we tested several configurations of JVM allowing it to exploit remote memory. Usually, Spark takes all the memory allocated to the JVM and fills it with data. Thus extra memory, which is slightly slower than local memory, will boost its performance until a level. To evaluate that level, we varied the amount of remote memory (up to 70 percent) in the JVM.

Fig. 10 shows the performance improvement while the amount of remote memory allocated to the VM is increasing. The graph shows that for all benchmarks, the performance is boosted until 40 percent of memory extension.

5 Related Work

The design of our system was inspired by Global Memory System [26] which was one of the first attempts to implement such an infrastructure-wide memory management service. As a memory management system which aims at memory mutualization, our contribution can be compared to related works within two categories:

1) Local Memory Mutualization: where the memory optimization and re-distribution is done within a single physical node.
2) Global Memory Mutualization: where the memory distribution is implemented datacenter-wide.



(a) Default implementation of Spark.

(b) Ohio Spark implementation.

Fig. 9. Performance evaluation with two different implementations of spark extended with remote memory.

Local Memory Mutualization.

Ginkgo [27] is a mutualization system which allows to dynamically adjust the memory size of a VM. It determines the minimum acceptable amount of memory that a VM may run with, based on application performance, memory usage, and submitted load. Then, it relies on ballooning for reclaiming unused memory. The main drawback of the method is that the user or the provider should profile the application in advance and inform the system with that profile.

Zhao and Wang [28] have introduced dynamic MEmory Balancer (MEB) for memory balancing between virtual machines. It estimates the memory consumption of a VM and periodically re-allocates memory of VMs based on their needs. In a more recent paper, they upgraded it with a non-intrusive working set estimation system [29]. This approach is close to our regarding local mutualization. However, they did not evaluate it with memory intensive applications such as Big Data applications.

Statistical resource multiplexing strategies have been introduced by Meng [30]. They analyze VMs' memory usage for deducing predictions and they create couples of VMs in such a way that when one reaches its peak memory consumption the second one is in low point of memory consumption. This forcasting of memory usage is complementary to our and could be inegrated in our system.

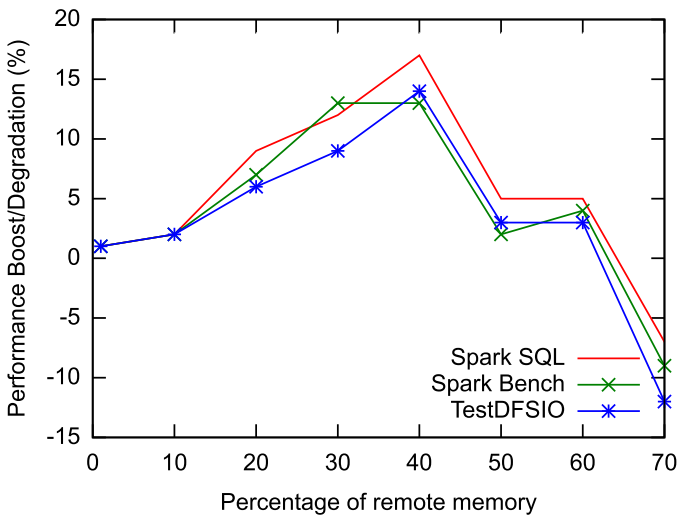


Fig. 10. Performance evaluation while increasing the allocated remote memory.

Zhang and al [31] present Memflex which is a shared memory swapper for improving guest swapping performance in virtualized environments. Our system allows a VM to use unused memory in the hole datacenter while Memflex is limited to the server hosting the VM. Therefore, we allow a better use of memory in the datacenter.

Global Memory Mutualization.

Remote swapping mechanisms [32], [33], [34], [35] were introduced years ago and deeply evaluated. However, as today's traditional Ethernet networks are not fast enough and remote paging causes a significant overhead on remote CPUs, successors have been implemented [36], [37] relying on low latency networking infrastructures (mainly Infiniband) enabling the use of RDMA technologies. Infiniswap [16] enables to create a swap device from the memory of one machine and to use that swap device from another machine through Infiniband. Infiniswap implements the fundamental mechanisms that we used to build a global memory mutualization system. [38] introduces PUMA which is a mechanism that improves I/O intensive applications performance by providing the ability for a VM to entrust clean page-cache pages to other VMs having unsused memory.

SpongeFiles [39] is a remote memory sharing systems for Hadoop. For large data sets generated by a job, it allows to avoid sending these data to disk but rather to route them to another node where sufficient memory is available. SpongeFiles addresses the issue of memory mutualization for Big Data application but limited to Hadoop applications. Our system addresses this issue in a virtualized infrastructures with a wider scope of applications.

Nswap2L [40] is a swap device extension for Linux which allows to add an abstraction level on swapping process. It appears to the OS as a single swap device partition whose data can be stored to or migrated between various heterogeneous storages (RAM, SSD, HDD etc.) including the memory of a

remote host. We share many objectives with Nswap2L, but our experiments target different environments. First, our system address memory mutualization in virtualized infrastructures. Second, while Nswap2L targets mutualization for HPC applications, we rather address mutualization between HPC clusters (with average memory consumption) and Big Data clusters (with high memory consumption). Finally, Nswap2L does not exploit RDMA based networking technologies.

Several research works use remote memory to implement a disaggregated architecture. [41] presents a remote memory system with remote servers being in a *Zombie* state. According to their nomenclature, a server is in the *Zombie* state if only its memory and its network interface is active, in order to reduce energy consumption. Other works such as [42] also offer operating systems for disaggregated architecture. These systems very often rely on the remote use of resources such as memory.

Position of Our Work.

The originality of our system compared to existing solutions comes from its completeness. It offers a system for (1) dynamically estimating working sets, (2) reserving remote memory (aggregate this memory into a distributed memory reservoir) and also (3) using this memory. Therefore, our solution proposes a overall system for improving HPC workload performance in a cluster by mutualizing memory usage. We offer a complete system that combines existing solutions with a clever management aspect, which is not done in existing works.

6 Conclusion

In this paper we present a platform which improves memory management of HPC and BD infrastructures via dynamically monitoring the working set of each VM, aggregating this memory into a distributed memory reservoir, and making it available to requiring VMs. Microbenchmarks, memory intensive benchmarks and Big Data benchmarks were used to evaluate our contribution. The results show that remote memory mutualization can improve the performance of a standard Spark benchmark by up 17 percent with an average performance degradation of 1.5 percent.

Acknowledgment

This work was benefited from the support of Région Occitanie under the Prématuration-2018 program, *Blablamem*.

References

[1] V. Infrastructure, "Resource management with VMware DRS," *VMware White Paper*, vol. 13, 2006.
[2] C. Subramanian, A. Vasan, and A. Sivasubramaniam, "Reducing data center power with server consolidation: Approximation and evaluation," in *Proc. Int. Conf. High Perform. Comput.*, 2010, pp. 1–10.
[3] R. R. Expósito, G. L. Taboada, S. Ramos, J. Touriño, and R. Doallo, "Performance analysis of HPC applications in the cloud," *Future Gener. Comput. Syst.*, vol. 29, no. 1, pp. 218–229, Jan. 2013. [Online]. Available: http://dx.doi.org/10.1016/j.future.2012.06.009
[4] C. P. Chen and C.-Y. Zhang, "Data-intensive applications, challenges, techniques and technologies: A survey on big data," *Inf. Sci.*, vol. 275, pp. 314–347, 2014.
[5] VMware, "Understanding memory resource management in VMwareESX server," *VMware White Paper*, 2009.
[6] A. Karve *et al.*, "Dynamic placement for clustered web applications," in *Proc. 15th Int. Conf. World Wide Web*, 2006, pp. 595–604. [Online]. Available: http://doi.acm.org/10.1145/1135777.1135865
[7] R. Linux, "SR-IOV." Accessed: Mar. 01, 2019. [Online]. Available: https://goo.gl/qUtsQz
[8] P. X. Gao *et al.*, "Network requirements for resource disaggregation," in *Proc. 12th USENIX Conf. Operating Syst. Des. Implementation*, 2016, pp. 249–264. [Online]. Available: http://dl.acm.org/citation.cfm?id=3026877.3026897
[9] M. Su, M. Zhang, K. Chen, Z. Guo, and Y. Wu, "RFP: When RPC is faster than server-bypass with RDMA," in *Proc. 12th Eur. Conf. Comput. Syst.*, 2017, pp. 1–15. [Online]. Available: http://doi.acm.org/10.1145/3064176.3064189
[10] V. Nitu, A. Kocharyan, H. Yaya, A. Tchana, D. Hagimont, and H. Astsatryan, "Working set size estimation techniques in virtualized environments: One size does not fit all," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 2, no. 1, pp. 19:1–19:22, Apr. 2018. [Online]. Available: http://doi.acm.org/10.1145/3179422
[11] C. A. Waldspurger, "Memory resource management in VMware ESX server," in *Proc. 5th Symp. Operating Syst. Des. Implementation*, 2002. [Online]. Available: http://www.usenix.org/events/osdi02/tech/waldspurger.html
[12] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Geiger: Monitoring the buffer cache in a virtual machine environment," in *Proc. 12th Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2006, pp. 14–24. [Online]. Available: http://doi.acm.org/10.1145/1168857.1168861
[13] P. Barham *et al.*, "Xen and the art of virtualization," in *Proc. 19th ACM Symp. Operating Syst. Princ.*, 2003, pp. 164–177. [Online]. Available: https://doi.org/10.1145/945445.945462
[14] A. Whitaker, M. Shaw, and S. D. Gribble, "Scale and performance in the Denali isolation kernel," in *Proc. 5th Symp. Operating Syst. Des. Implementation*, 2002. [Online]. Available: http://www.usenix.org/events/osdi02/tech/whitaker.html
[15] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed prefetching and caching," in *Proc. 15th ACM Symp. Operating Syst. Princ.*, 1995, pp. 79–95. [Online]. Available: https://doi.org/10.1145/224056.224064
[16] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with INFINISWAP," in *Proc. 14th USENIX Symp. Netw. Syst. Des. Implementation*, 2017, pp. 649–667. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu
[17] M. Ferdman *et al.*, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *Proc. 17th Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2012. [Online]. Available: http://infoscience.epfl.ch/record/173764
[18] E. Benchmarks, "Elasticsearch nightly benchmarks." Accessed: Mar. 01, 2019. [Online]. Available: https://elasticsearch-benchmarks.elastic.co/
[19] A. Ghazal *et al.*, "BigBench V2: The new and improved BigBench," in *Proc. IEEE 33rd Int. Conf. Data Eng.*, 2017, pp. 1225–1236.
[20] M. Armbrust *et al.*, "Spark SQL: Relational data processing in spark," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 1383–1394. [Online]. Available: http://doi.acm.org/10.1145/2723372.2742797
[21] S. Benchmarks, "TestDFSIO spark benchmark." Accessed: Mar. 01, 2019. [Online]. Available: https://github.com/BBVA/spark-benchmarks
[22] 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. Front.*, 2015, pp. 53:1–53:8. [Online]. Available: http://doi.acm.org/10.1145/2742854.2747283
[23] P. Luszczek, E. Meek, S. Moore, D. Terpstra, V. M. Weaver, and J. Dongarra, "Evaluation of the HPC challenge benchmarks in virtualized environments," in *Proc. Int. Conf. Parallel Process.*, 2012, pp. 436–445. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-29740-3_49
[24] X. Lu, M. W. U. Rahman, N. Islam, D. Shankar, and D. K. Panda, "Accelerating spark with RDMA for big data processing: Early experiences," in *Proc. IEEE 22nd Annu. Symp. High-Perform. Interconnects*, 2014, pp. 9–16. [Online]. Available: http://dx.doi.org/10.1109/HOTI.2014.15
[25] Databricks, "Join performance workload for spark SQL." Accessed: Mar. 01, 2019. [Online]. Available: https://github.com/databricks/spark-sql-perf
[26] M. J. Feeley, W. E. Morgan, E. P. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath, "Implementing global memory management in a workstation cluster," in *Proc. 15th ACM Symp. Operating Syst. Princ.*, 1995, pp. 201–212. [Online]. Available: http://doi.acm.org/10.1145/224056.224072

[27] 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.
[28] W. Zhao and Z. Wang, "Dynamic memory balancing for virtual machines," in *Proc. 5th Int. Conf. Virt. Execution Environ.*, 2009, pp. 21–30. [Online]. Available: https://doi.org/10.1145/1508293.1508297
[29] Z. Wang, X. Wang, F. Hou, Y. Luo, and Z. Wang, "Dynamic memory balancing for virtualization," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 1, pp. 2:1–2:25, Mar. 2016. [Online]. Available: http://doi.acm.org/10.1145/2851501
[30] X. Meng, C. Isci, J. Kephart, L. Zhang, E. Bouillet, and D. Pendarakis, "Efficient resource provisioning in compute clouds via VM multiplexing," in *Proc. 7th Int. Conf. Autonomic Comput.*, 2010, pp. 11–20. [Online]. Available: http://doi.acm.org/10.1145/1809049.1809052
[31] Q. Zhang, L. Liu, G. Su, and A. Iyengar, "MemFlex: A shared memory swapper for high performance VM execution," *IEEE Trans. Comput.*, vol. 66, no. 9, pp. 1645–1652, Sep. 2017.
[32] T. E. Anderson, D. E. Culler, and D. A. Patterson, "A case for NOW (networks of workstations)," *IEEE Micro*, vol. 15, no. 1, pp. 54–64, Feb. 1995. [Online]. Available: https://doi.org/10.1109/40.342018
[33] E. P. Markatos and G. Dramitinos, "Implementation of a reliable remote memory pager," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf.*, 1996, pp. 15–15. [Online]. Available: http://dl.acm.org/citation.cfm?id=1268299.1268314
[34] S. Dwarkadas, N. Hardavellas, L. Kontothanassis, R. Nikhil, and R. Stets, "Cashmere-VLM: Remote memory paging for software distributed shared memory," in *Proc. 13th Int. Parallel Process. Symp. 10th Symp. Parallel Distrib. Process.*, 1999, pp. 153–159.
[35] D. Williams, H. Jamjoom, Y.-H. Liu, and H. Weatherspoon, "Overdriver: Handling memory overload in an oversubscribed cloud," in *Proc. 7th Int. Conf. Virt. Execution Environ.*, 2011, pp. 205–216. [Online]. Available: https://doi.org/10.1145/1952682.1952709
[36] H.-H. Choi, K. Kim, and D.-J. Kang, "Performance evaluation of a remote block device with high-speed cluster interconnects," in *Proc. 8th Int. Conf. Comput. Model. Simul.*, 2017, pp. 84–88. [Online]. Available: http://doi.acm.org/10.1145/3036331.3050420
[37] 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.
[38] M. Lorrillere, J. Sopena, S. Monnet, and P. Sens, "Puma: Pooling unused memory in virtual machines for I/O intensive applications," in *Proc. 8th ACM Int. Syst. Storage Conf.*, 2015. [Online]. Available: https://doi.org/10.1145/2757667.2757669
[39] K. Elmeleegy, C. Olston, and B. Reed, "SpongeFiles: Mitigating data skew in MapReduce using distributed memory," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 551–562. [Online]. Available: http://doi.acm.org/10.1145/2588555.2595634
[40] T. Newhall, E. R. Lehman-Borer, and B. Marks, "Nswap2L: Transparently managing heterogeneous cluster storage resources for fast swapping," in *Proc. 2nd Int. Symp. Memory Syst.*, 2016, pp. 50–61. [Online]. Available: http://doi.acm.org/10.1145/2989081.2989107
[41] V. Nitu, B. Teabe, A. Tchana, C. Isci, and D. Hagimont, "Welcome to Zombieland: Practical and energy-efficient memory disaggregation in a datacenter," in *Proc. 13th EuroSys Conf.*, 2018. [Online]. Available: https://doi.org/10.1145/3190508.3190537
[42] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "LegoOS: A disseminated, distributed OS for hardware resource disaggregation," in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 69–87. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/shan

Aram Kocharyan received the PhD degree from the Polytechnic National Institute of Toulouse, Toulouse, France, and the Institute for Informatics and Automation Problems, National Academy of Sciences of Armenia, Yerevan, Armenia, in 2019. His main research interests include virtualization, cloud computing, and operating system.

Brice Ekane received the MS degree from the University of Yaounde, Yaounde, Cameroon, in 2011. He is currently working toward the PhD degree with the IRIT Lab, Toulouse, France, since September 2018. He is a member of SEPIA Research Group. His main research interests include virtualization, cloud computing, and operating system.

Boris Teabe received the PhD degree from the Polytechnic National Institute of Toulouse, Toulouse, France, in 2017. Since 2019, he is an assistant professor with INP Toulouse, Toulouse, France. His main research interests include virtualization, cloud computing, and operating system.

Giang Son Tran received the PhD degree in computer science from the IRIT Laboratory, Polytechnic National Institute of Toulouse, Toulouse, France, in 2014. Since August 2014, he is a lecturer with the University of Science and Technology of Hanoi and a researcher with ICTLab. His research interests include operating system, mobile platforms, machine learning, and high performance computing.

Hrachya Astsatryan received the PhD degree from the Institute for Informatics and Automation Problems, National Academy of Sciences of Armenia, Yerevan, Armenia, in 2001. Since 2014, he is head of the Center for Scientific Computing, National Academy of Sciences of Armenia.

Daniel Hagimont received the PhD degree from the Polytechnic National Institute of Grenoble, Grenoble, France, in 1993. He is a professor with the Polytechnic National Institute of Toulouse, France and a member of the IRIT Laboratory, where he leads a group working on operating systems, distributed systems, and middleware. After a postdoctorate with the University of British Columbia, Vancouver, Canada, in 1994, he joined INRIA Grenoble, in 1995. He took his position of professor in Toulouse, in 2005.