Shared Memory Optimization in Virtualized Cloud

Qi Zhang and Ling Liu
Georgia Institute of Technology, Atlanta, Georgia, USA
{*qzhang90, lingliu*}*@cc.gatech.edu*

***Abstract*—Shared memory management is widely recognized as an optimization technique in the virtualized cloud. Most current shared memory techniques allocate shared memory resources from guest VMs based on pre-defined system configurations. Such static management of shared memory not only increases the VM memory pressure, but also limits the flexibility to balance the shared memory resources across multiple VMs running on a single host. In this paper, we present a dynamic shared memory management framework which enables multiple VMs to dynamically access the shared memory resource according to their demands. We illustrate our system design through two case studies: one aims at improving the performance of inter-domain communication while the other aims at improving VM memory swapping efficiency. We demonstrate that the dynamic shared memory mechanism not only improves the utilization of shared memory resources but also significantly enhances the performance of VM applications. Our experimental results show that by using dynamic shared memory management, we can improve the performance of inter-VM communication by up to 45 times, while mitigating the VM memory swapping overhead by up to 58%.**

***Keywords*-Shared memory, resource management, virtualization.**

I. Introduction

Shared memory technique, originally introduced for optimizing inter process communication, is gaining increasing attraction as a kernel-level optimization technique for efficient executions of virtual machines (VMs) in virtualized cloud and data centers. For example, instead of communicating through the host OS or hypervisor, shared memory channel established between co-located VMs may enable network packets to be transmitted from one VM to the other without the involvement of the host OS or the hypervisor coordination [11–13, 15, 17]. By shortening the inter-VM communication path through shared memory, we can significantly enhance the communication performance for both TCP and UDP workloads. Another example is to utilize shared memory between VMs to achieve efficient VM memory swapping. Concretely, when a VM is under high memory pressure and needs to swap some pages out, instead of swapping pages to disk, pages can be swapped to the other VM's memory which is shared to this VM. Despite the benefits of shared memory in virtualized cloud, most of current shared memory optimizations allocate shared memory resources from guest VMs at system configuration time. Such static management of shared memory not only increases the VM memory pressure in the presence of memory intensive workloads, but also significantly limits the flexibility to balance the shared memory resources across multiple VMs running on a single host, especially when the shared memory demands from different VMs vary from workload to workload. Challenges still exist in terms of how to manage the shared memory resource in a more efficient manner while enabling the applications in VMs to keep being benefited from the shared memory technique.

Problems in static management of shared memory. Most of the existing shared memory methods deployed in a virtualized cloud statically allocate the shared memory between a pair of communicating VMs [6]. This rigid management of shared memory suffers from the inflexibility in the presence of highly skewed memory intensive workload distribution across multiple co-located VMs. Therefore, the static shared memory management in virtualized cloud may result in either resource waste or VM performance degradation. In the inter domain communication cases, for instance, when a sender VM tries to forward some network packets to its co-located receiver VM, it will allocate a piece of memory from its own main memory as the communication channel with the co-located receiver VM in order to avoid the involvement of host or hypervisor during the packets transmission. However, the size of this shared memory cannot be adaptively adjusted based on the intensity of the network traffics. If the size of the allocated shared memory is much larger than the workloads' actual requirement, the under utilized shared memory resource will be wasted. On the other hand, if the allocated shared memory is insufficient to meet the workload demands, the performance of the application running on the VMs will be affected [17].

Balance of shared memory across VMs. Another problem inherent in many current shared memory management schemes is related to the system software stack where the shared memory should be allocated. Existing shared memory mechanisms in virtualized cloud typically provide allocation of shared memory from guest VMs. In other words, a VM needs to spare some of its own memory resource in order to provide shared memory with other VMs. This guest VM based shared memory allocation, although improves the communication performance by using shared memory, may cause performance degradation for memory intensive applications running on the VMs due to frequent memory page swapping activities caused by insufficient working memory with respect to the application.

In this paper we address the above mentioned problems by promoting a dynamic shared memory management framework for improving virtual machine execution efficiency in virtualized cloud. Our framework has a number of unique features. First, we advocate to slice the shared memory region allocated to each guest VM into small memory chunks and utilizing the demand paging concept such that shared memory is allocated one chunk at a time. Second, instead of allocating shared memory from inside of each guest VM, we advocate the host-based (or hypervisor-level) shared memory allocation scheme, aiming at increasing the flexibility of allocating and de-allocating shared memory across multiple co-located VMs according to their changing demands. Both techniques enable the flexible sharing and well-balanced management of shared memory resources across VMs on a single host. Thus, the utilization of shared memory resources can be significantly improved by just in time allocation of the shared memory to the VMs that need it. At the same time, by pooling the shared memory resources and managing them at the host OS or hypervisor, shared memory for each guest VM can be enlarged or shrank on demand with minimal performance overhead. To evaluate the effectiveness of our dynamic shared memory management design, we develop two prototype systems *MemPipe* and *MemFlex* to show how we use dynamic shared memory management mechanism to improve the inter-VM communication performance and mitigate the performance interference introduced by VM memory swap respectively. Our experimental evaluation results using these two prototype systems demonstrate that our dynamic shared memory management framework can significantly improve the execution performance of applications running on VMs, while achieving efficient utilization of shared memory resources.

II. Related Work

The shared memory techniques have been adopted in distributed systems and virtualized cloud and data centers. TreadMarks [5] proposed a distributed shared memory system enables to processes running on different physical nodes to access a globally shared virtual memory address. Similarly, [10] presents the research effort for building an elastic operating system for datacenter. The main idea is to eliminate the boundaries among the physical nodes in the same cluster, and enable the process running on one machine to access the memory resources on the other machines. In addition, a new file system was designed in [8] to promote the disk I/O throughput by phase change memory. Efforts for improving network I/O performance, such as [17] [11] [12], and [13] have also been made to accelerate the inter-VM communication performance by establishing shared memory channel among communicating VMs. In the context of VM memory swapping, [4] addressed the double paging problem in virtualized cloud by introducing cooperative memory swapping between the host and the VMs. [18] swaps the VM memory memory to another remote VM through network virtualization, in order to alleviate the performance impact brought by VM memory swapping in a oversubscribed virtualized cloud. Although swapping through network can be much slower than through shared memory, it can be a viable alternative when disk I/O bandwidth is significantly worse than the network I/O bandwidth.

To the best of our knowledge, this paper is the first effort on desigining and implementing a *dynamic* shared memory management framework, for improving virtual machine execution efficiency by optimizing both inter-VM communication performance and VM memory swapping performance.

III. Dynamic Management of Shared Memory

In this section, we describe the design of our dynamic shared memory management framework. In the first two subsections, we introduce guest-VM based and host-OS or hypervisor based shared memory allocation methods. We will describe the shared memory allocation mechanism within a guest-VM using the grant table, a commonly used kernel interface for memory sharing in virtualization platforms. We also discuss the limitation of grant table based shared memory management, and describe a host-based dynamic shared memory mechanism that we have designed and implemented. In the next two subsections, we present two case studies: inter domain network communication and virtual machine memory swapping, to demonstrate the benefit of our host-based dynamic shared memory management for virtualized cloud.

A. Shared Memory Allocation in Guest VMs

The Grant table mechanism is widely used in virtualization platforms, such as Xen [6], to enable memory pages being shared between VMs running on the same physical machine. In Xen, each VM is initialized with a set of pages shared with the hypervisor, and the grant table of each VM is located in these shared pages. The entries in the grant table are used to locate shared memory pages between two VMs. For instance, if *VM1* wants to share one of its memory page $P1$ with *VM2*, it first fills in one of the entries in its grant table with the property values such as the page frame to be shared, the destination VM, the access permissions of the page, and so forth. Then, *VM1* transmits the reference of the grant table entry to *VM2*. Upon receiving the shared memory reference, *VM2* uses it to map the granted page frame into its own memory address space, for example by calling *HYPERVISOR_grant_table_op()* with parameter *NTTABOP_map_grant_ref* in Xen. After performing the memory access on the shared page frame, *VM2* unmaps the granted page to free the shared memory page frame, for example, by calling *HYPERVISOR_grant_table_op()* but with the parameter *GNTTABOP_unmap_grant_ref* in Xen. Finally, *VM1* removes the entry from the grant table such

that the specific page frame allocated as shared memory region is exclusively owned by *VM1* again.

Although the grant table mechanism provides generic interfaces for convenient memory sharing between VMs, it suffers from a number of limitations. For example, the shared memory is allocated statically. Thus, allocating a large shared memory region may waste memory resources that are needed by other applications running on the same guest VMs. On the other hand, allocating a too small shared memory region may degrade the performance when it is insufficient to satisfy the demands from workloads.

B. Shared Memory Allocation in Host/Hypervisor

An alternative way to establish shared memory across multiple co-located VMs in virtualized platform is to allocate a global memory region from the host. This shared memory region is dynamically managed such that each VM is allowed to access the shared memory based on its workload requirements. Unlike the grant table service provided by the guest OS kernel, there is no existing interface allowing VMs to dynamically and proportionally access the memory region allocated by the host in a shared manner. Therefore, we first discuss how to establish the shared memory among host and VMs. Then, we illustrate our dynamic shared memory management framework by using two concrete case studies: inter domain communication optimization and virtual machine memory swapping optimization.

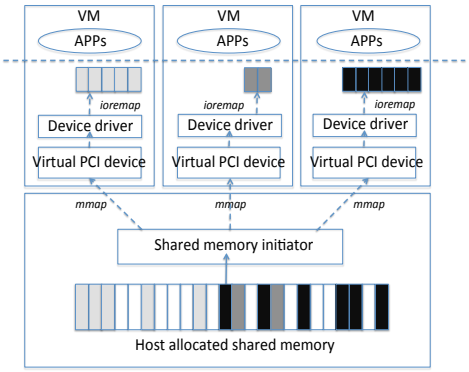


Figure 1. Host allocated dynamic shared memory

As shown in Figure 1, the host based shared memory allocation is achieved through PCI device I/O memory remap. A shared memory initiator is created on the host, which is responsible for allocating a piece of memory region via *shm_open()* from the host OS. *shm_open()* creates and opens a new POSIX shared memory object, and returns the handler of the shared memory object, which can be used by other processes to map into their own address spaces. At the same time, a virtual PCI device is created and assigned to each guest VM. After the shared memory object is created, the PCI device creates a BAR (Base Address Register) in its configuration space to hold the object. It is worth noting that, in order to allow BAR to point to a memory region larger than 1GB, the *PCI_BASE_ADDRESS_MEM_TYPE_64* bit must be set when the BAR of the virtual PCI device is registered. In addition to the virtual PCI device, each guest VM is assigned to a PCI device driver. The driver maps the memory region indicated by the BAR of the virtual PCI device into the guest VM's kernel address space by invoking *ioremap_cache()*, which returns the kernel virtual address of the shared memory region after being mapped. Although it is quite possible that these kernel virtual addresses are different in different VMs, as long as the mapping is successful, the VMs are able to access the same memory region allocated from the host. Therefore, when multiple co-located VMs map the host memory region into their kernel address spaces simultaneously, the memory region allocated from the host becomes the shared memory among these guest VMs.

In addition to establishing a host allocated global shared memory among guest VMs, dynamic shared memory management is another key challenge that should be addressed in order to better capitalize on the potential benefits of shared memory in virtualized cloud. In the next two subsections, we present the key techniques for designing a dynamic shared memory management framework through two case studies and discuss why dynamic shared memory management is essential in virtualized cloud.

C. Case study: Inter Domain Communication

Network I/O workloads among VMs are dominating in most of the virtualized cloud today. For example, many big data processing frameworks such, as MapRedce [9] and Spark [2], require intermediate data to be shuffled around the worker nodes. The shuffle phase often becomes the bottleneck of the whole big data processing job due to the massive network I/Os among VMs [3]. Efforts such as using infiniband and RDMA (Remote Direct Memory Access) [14] have been made to improve the network performance among physical machines. While shared memory mechanisms are proposed to accelerate the communication efficiency among co-located VMs [11–13, 17]. However, an inherent problem that exists in most of the existing shared memory inter-VM communication systems is the inflexibility of shared memory management, which usually results in unbalanced shared memory resource utilization and degradation of the inter-VM communication performance.

Concretely, the first problem inherent in most shared memory inter-VM communication systems is the static shared memory management. Namely, the size of shared memory channel used for co-located VM communication is set at the system configuration time and fixed once it is initialized. The second problem is about what is the right size of shared memory. Since the network I/O workloads in VMs vary from time to time, it is unreasonable to use a fixed size shared memory channel for co-located inter-VM communication. Furthermore, even if the network workloads

in the VMs are stable, it is difficult to decide the appropriate shared memory size that will be required by the workloads prior to their execution. Thus, the size of shared memory in existing methods is often arbitrarily decided without systematic guidance.

Therefore, we argue that the shared memory channel for inter-VM communication needs to be managed in a dynamic and workload adaptive manner to achieve both efficient memory resource utilization and high inter-VM communication performance [19]. The host based shared memory allocation provides a good opportunity to achieve flexible shared memory management in a virtualized cloud.

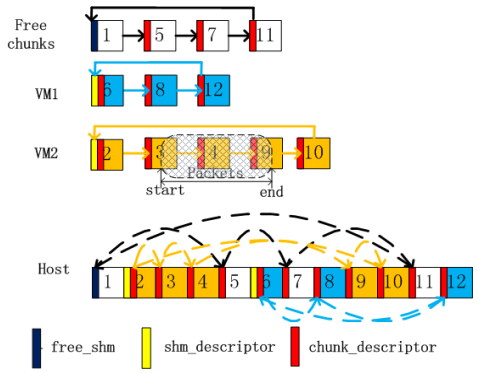


Figure 2. Shared memory organization

We propose a dynamic shared memory management framework with three novel features: (1) Partitioning the shared memory resource into smaller memory chunks; (2) Employing the on demand principle to allocate each guest VM shared memory chunks proportional to its workload demands; and (3) Flexible management of free shared memory chunks to provide demand-utility balanced shared memory allocation. Figure 2 describes some details about how the memory region allocated from the host is dynamically shared by multiple VMs. First, we organize the shared memory region in terms of small memory chunks, the size of each chunk can be specified in the configuration file. Each shared memory chunk serves as the basic unit in dynamic shared memory allocation and de-allocation operations. Second, each shared memory channel between a sender VM and a co-located receiver VM is initialized with a single shared memory chunk. Additional memory chunks will be added to the shared memory channel proportionally to the demand of the network workloads between the two VMs. Memory chunks belonging to the same shared memory channel are organized into a linked list. Meanwhile, all the free chunks, which do not belong to any allocated shared memory channel, are also organized into a linked list of free chunks.

An important goal for dynamic shared memory management is to enable multiple shared memory channels for different pairs of inter-VM communication co-exist in the same host memory region, such that the dynamic expanding and shrinking of one shared memory channel does not affect the allocation and de-allocation of other shared memory channels. We introduce three metadata structures: *free_shm*, *shm_descriptor*, and *chunk_descriptor*.

The first metadata structure *free_shm* is a global metadata, which locates in the very beginning of the shared memory and is accessed by all the VMs that using the shared memory channel. The second metadata structure is *shm_descriptor*, which records the metadata for each single shared memory channel. The key properties in this metadata are two pairs of pointers. The first pair is *front, back*, which points to the beginning and the end of the area where the data packets are occupied. The values of *front* and *back* are in terms of the offset in the global allocated shared memory region. Given that the shared memory region is organized in memory chunks, it is quite possible that the packet size is not aligned with the chunk size. Therefore, the second pair of pointers, *front_mdata* and *back_mdata* are used to specify the two memory chunks in which the *front* and the *back* are located respectively. The third metadata structure is *chunk_descriptor*, which maintains the metadata of each single memory chunk. It contains *prev* and *next*, which act as the pointers to link different memory chunks into a list.

Each shared memory channel is initialized with a single memory chunk. Based on the workload demands, each VM can require more allocations of shared memory chunks for its shared memory channel. Thus the size of different shared memory channels may vary based on the workload demands from each of the guest VMs. We achieve this dynamic allocation by creating a separate thread, which monitors the utilization of each shared memory channel and dynamically adjusts the channel capacity.

The process of setting up a shared memory channel involves three handshakes. Specifically, when a sender VM (e.g. VM1) wants to send a packets to its co-located receiver VM (e.g., VM2), a *channel create* message is generated by VM1 and sent to VM2. Upon receiving this message, VM2 initialize a shared memory channel from the host allocated memory region, and sends a *create ACK* message, which includes the handler of the newly initialized shared memory channel, back to VM1. After receiving the *create ACK*, VM1 extract the shared memory channel handler from it and connects itself to the channel. Then, VM1 sends a *create FIN* message to VM2 to confirm that it has been successfully connected to the shared memory channel. From this point on, VM1 is able to deliver packets to VM2 through the established shared memory channel.

D. Case Study: Virtual Machine Memory Swapping

Virtual machine memory swapping is another case study that we have performed in terms of implementing and testing the shared memory management framework to show the benefits of dynamic shared memory management in virtualized cloud.

It is widely recognized that the number of concurrently running VMs on a single host machine depends on the capacity of hardware resources and the efficiency of hardware overcommitment technologies. In order to increase the density of VMs on the host platform, lots of efforts have been made on developing innovative schedulers for CPU to provide more adaptive CPU resources sharing among VMs. However, memory resource remains to be the bottleneck in virtualized environment. Unlike CPU, which can be shared by time slicing, memory resource is shared by space slicing and moving memory pages from one VM to another is relatively expensive.

Balloon driver [16] is a commonly used approach to enable memory resource overcommitment in virtualized cloud. However, ballooning driver is not a panacea. For instance, it may not move memory fast enough to satisfy the memory requirement of some VM, since VM memory swapping could be triggered while balloon driver is moving the memory across guest VMs. Another issue is the double paging problem, which is caused by the uncooperative swapping activities between the guest OSs and the host OS. The unnecessary disk I/O introduced by double paging problem could seriously affect the execution performance of guest VMs.

We argue that utilizing shared memory in the host can be an attractive system optimization for mitigating the VM performance degradation brought by VM memory swapping, while alleviating the double paging problem. From VM perspective, if there is spare host memory, then the capability of utilizing the host memory as swap area instead of disk would always be more efficient because the access speed of disk is usually about five magnitudes slower than that of memory [20].

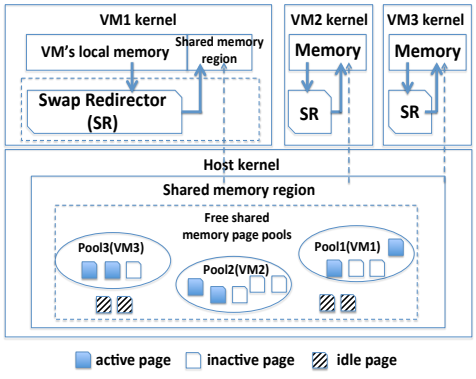


Figure 3. VM memory swapping via dynamic shared memory

Figure 3 gives a sketch of VM memory swapping through shared memory. First, a memory region is allocated in the host as the VM swapping area. Also, the memory pages in the shared memory region are partitioned into multiple pools, and each pool is assigned to a VM as its swap partition. Second, this globally allocated host memory region is mapped to the guest kernel space of each VM. Similar to the shared memory design for the inter domain communication optimization case, the shared memory pages belonging to the same pool are organized into a linked list, and each pool is initialized with a preconfigured size. A hash table is maintained to record the VM id and its corresponding pool address. Each VM needs to refer to this hash table to find where to swap their memory pages. A separate thread is created to monitor the utilization of each shared memory pool, and dynamically adjust the size of pools based on the memory swapping traffics from the VMs. If all the pages in a shared memory pool are used and no more free pages in the globally allocated shared memory region is available, the swapping traffics are automatically resort to the disk partition.

By supporting shared memory swapping, we can replace the disk I/O operations performed by conventional memory swapping by the copy from VM memory to host shared memory region, which significantly reduces the latency of memory swapping operations and mitigates the performance interference caused by VM memory swapping. Shared memory swapping has a number of advantages. First, it alleviates the performance degradation of the applications running on the same VM where frequent memory swapping occurs. Second, it benefits the other co-located VMs that have disk I/O intensive workloads, because the host disk I/O bandwidth can be allocated to those VMs that are running disk I/O intensive workloads. Third but not the least, shared memory based approach improves the swapping efficiency and mitigates the double paging problem. Thus, we argue that shared memory based VM memory swapping can be an attractive performance optimization mechanism when some of the host memory is available and can be utilized as shared memory region.

IV. Implementation

We implement our dynamic shared memory framework on KVM platform, and build a set of interfaces for VM kernel to access the shared memory region. For example, *struct shm_area* shm_init(size_t size)* and *void shm_exit(struct shm_area)* are two interfaces for VM kernel to allocate and destroy the shared memory region from the host. The return value *struct shm_area** include the offset and the length of the allocated shared memory. Access control is another category of functionality that can be incorporated into our system. Based on the dynamic shared memory management interfaces, we build two proof-of-concept systems: *MemPipe*, which is a inter domain communication system, and *MemFlex*, which is a shared memory swapping system.

V. Evaluation

The evaluation is carried out in three steps. First, we measure the performance overhead of dynamic shared memory management by comparing the performance of benchmarks

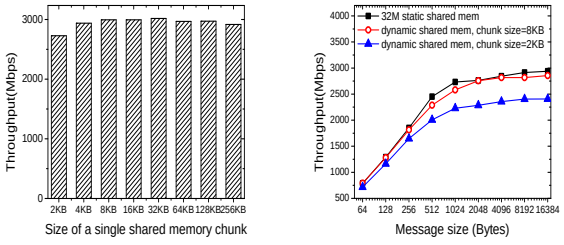
running under dynamic and static shared memory management. Second, we evaluate how the performance of inter domain communication can be improved by utilizing dynamic managed shared memory channels. Third, we demonstrate the effectiveness of shared memory swapping.

Our experimental environment includes two physical machines. Each machine has a four core 2.4GHz Intel CPU, 4GB memory, and 500GB disk, and an Gigabit network interface card. KVM 3.6 with QEMU 1.2.0 is installed in the host machine, and Linux kernel 2.6.34.14 are used as the operating system of both host and VMs. The benchmarks used include: Netperf [1], Dacapo [7], MapReduce jobs, and other network applications such as scope, wget, sftp, and etc.

A. Dynamic vs. Static Shared Memory Management

Compared with static shared memory management, in which a single piece of shared memory is allocated to the VM at the beginning of the shared memory channel setup, dynamic shared memory management requires memory chunks to be added to and removed from the shared memory channels based on the changing demands from the VM workloads. In this set of experiments, we vary the shared memory chunk size from 2KB to 256KB to investigate how the benchmark performs in different chunk sizes. The throughput of inter domain communication is used as the performance metric.

Figure 4(a) shows the throughput measured by TCP_STREAM workload generated by Netperf. The size of the messages in the workload is 8KB, and all the throughput are measured under the circumstance of dynamic shared memory management. We observe that when the chunk size increases from 2KB to 8KB, the achieved throughput slightly increases by 12.4% from 2728 Mbps to 3066 Mbps. However, the throughput does not further increase when the chunk size is larger than 8KB. This indicates that when the memory chunks size is smaller than 8KB, the high frequency of memory chunk allocation and deallocation degrades the performance of the benchmark. When the memory chunk size becomes no less than 8KB, the frequency of memory chunk allocation and deallocation during the workload execution becomes lower, and its impact on the benchmark performances becomes negligible.



(a) Performance vs. chunk size (b) Static vs. dynamic shared mem

Figure 4. Performance overhead of shared memory management

A smaller chunk size enables more flexible shared memory management, while a larger chunk size introduces less performance overhead. Results from Figure 4(a) give a hint that 8KB may be a reasonable choice for the chunk size to tradeoff between the flexibility and performance overhead. However, the message size in the workload used in figure 4(a) is also 8KB, and we need to make sure whether 8KB chunk size also fits other workloads with different message sizes. Therefore, we compare the performance of workloads with different message size when they are running under static and dynamic shared memory management. We vary the message size of workloads from 64B to 16KB, and allocate 32MB shared memory as the inter domain communication channel in static scenario, which is big enough to guarantee that no packets will be dropped due to the insufficient shared memory channel space. Figure 4(b) shows that no matter for what message size, the inter domain communication performance in dynamic shared memory management case with 8KB chunk size is very similar to those in static shared memory case. Therefore, we choose 8KB as the chunk size in dynamic shared memory management for the following experiments.

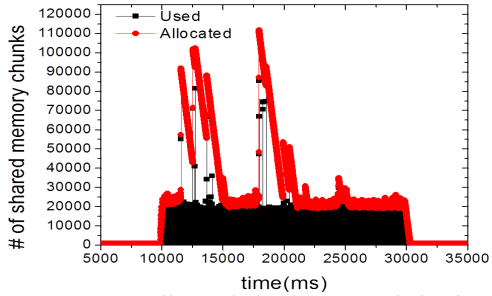


Figure 5. Allocated chunks vs. used chunks

B. Inter Domain Communication

This set of experiments uses MemPipe to show how the shared memory resource is allocated with the varying demand from the VM. We also demonstrate the inter domain communication performance improvement by adopting the dynamic managed shared memory channels.

Figure 5 illustrates the allocated and utilized shared memory resource under dynamic shared memory management when a UDP_STREAM workload is running between two co-located VMs. The red line represents the number of allocated shared memory chunks while the black line shows the amount of actually used shared memory chunks over the measurement period.

Table I displays the shared memory utilization when a UDP_STREAM workload generated by Netperf is running between two co-located VMs. In static shared memory allocation case, since it is hard to accurately infer an appropriate size of shared shared memory for workloads of different message sizes, we experimentally allocate 1MB as the shared memory channel between the 2 VMs. While in the dynamic case, the shared memory is allocated based on

Table I
COMPARISON OF SHARED MEMORY UTILIZATION, **udp sender**

msg size (Byte)	Static (Average values)				Dynamic (Average values)			
	Allocated(KB)	Used(KB)	Util	Throughput (Mbps)	Allocated(KB)	Used(KB)	Util	Throughput (Mbps)
64	1024	6	**0.5%**	128	14	6	**45%**	124
128	1024	11	**1.0%**	258	18	10	**57%**	246
256	1024	17	**1.6%**	503	26	17	**66%**	495
512	1024	31	**3.0%**	1005	42	31	**73%**	947
1024	1024	55	**5.2%**	1790	70	56	**80%**	1755
2048	1024	100	**9.5%**	2846	124	108	**87%**	2815
4096	1024	216	**20.6%**	4570	227	205	**90%**	4516
8192	1024	409	**39.0%**	5233	451	415	**92%**	5198
16384	1024	823	**78.5%**	6006	814	814	**94%**	5904

Table II
INTER VM COMM. PERFORMANCE MEASURED BY NETSURF

	msg size	Inter Machine	Inter VM	MemPipe
TCP_STREAM (Mbps)	1 KB	937	135	2579
	4 KB	936	137	2818
UDP_STREAM (Mbps)	1 KB	932	58	1755
	4 KB	950	99	4517
TCP_RR (#trans/sec)	1 KB	4845	2542	6265
	4 KB	4040	1412	4421
UDP_RR (#trans/sec)	1 KB	4968	2589	6830
	4 KB	4067	1468	6262

the demands of the UDP_STREAM workload. There are two observations from this table. First, since the workload demand changes with difference message size, the static shared memory allocation results in low resource utilization. For example, when the message size is 64 bytes, only 0.5% of shared memory has been used in static case. However, dynamic shared memory mechanism can adaptively adjust the shared memory size based on the workload demands, and achieve a higher resource utilization, which is between 45% and 94%. Second, by comparing the throughput in static and dynamic cases, we find that dynamic shared memory management leads to only up to 5.8% performance overhead.

Table II compares the performance of inter domain communication in three cases: *Inter Machine*, which refers to communication between two physical machines, *Inter VM* and *MemPipe*, which represent the communication throughput between two co-located VMs through native network stack and dynamic managed shared memory channel respectively. TCP_STREAM, UDP_STREAM, TCP_RR, and UDP_RR workloads generated by Netperf are used in this experiment. We find that no matter for which workload, MemPipe always outperforms the other two cases. For instance, when the message size is 4KB, the throughput of TCP_STREAM workload reaches 2818Mbps in MemPipe case, which is 3.01 times and 20.60 times higher than that in Inter Machine case and Inter VM case respectively. Also, the performance gap between MemPipe and the other two cases becomes bigger when the message size in the streaming workload increases from 1KB to 4KB. Taking UDP_STREAM workload for an example, MemPipe outperforms Inter Machine and Inter VM case 1.88 times and 30.26 times respectively when the message size is 1KB, and the corresponding numbers grow up to 4.75 and 65.46 when the message size increases to 4KB. This is because

Table III
PERFORMANCE OF NETWORK APPS

	scp	wget	vsftp-put	vspft-get	sftp-put	spft-get
Baseline(MB/s)	12	11	14	15	17	18
MemPipe(MB/s)	22	35	26	27	33	33

Table IV
EXECUTION TIME OF DACAPO BENCHMARKS

		eclipse	h2	jython	xalan	total
VM1(ms)	Baseline	214224	44408	17292	4555	280479
	MemFlex	76922	24814	13045	3945	118726
VM2(ms)	Baseline	168505	36390	14498	3952	223345
	MemFlex	73147	12842	14013	3851	103853

communication with larger message size is more network bounded while network workloads with smaller message size is more CPU bounded, and network bounded workloads can be benefited from communicating via dynamic shared memory channel by shipping packets through a shorter path.

Besides using the benchmarks, we also measure the performance of some widely used network applications by using MemPipe. Table III shows that compare with the Inter VM case, MemPipe improves the performance of these applications from 1.80 times to 3.18 times. For example, the throughput of vsftp-get has been improved 15MB/s to 27MB/s, while that of wget increases more than 3 times.

C. Shared Memory Based VM Memory Swapping

We evaluate the performance of MemFlex on two VMs (i.e., VM1 and VM2) running on the same host, and workloads from Dacapo are executing sequentially on the two VMs. Since some workloads from Dacapo, such as *h2* and *eclipse*, are memory intensive, while others, such as *jython* and *xalan*, are CPU intensive. We vary the execution order of workloads in different VMs to create a scenario in which the memory demand changes from time to time in each VM. Concretely, the workloads executing in VM1 are ordered as following: $jython \rightarrow h2 \rightarrow xalan \rightarrow eclipse$, while that in VM2 are arranged as $eclipse \rightarrow xalan \rightarrow jython \rightarrow h2$.

Table IV shows the execution time of each workload as well as the total execution time of all the workloads from the two VMs. We use baseline to denote the conventional VM memory swap without turning on MemFlex. We make several interesting observations. *First*, the total execution time of the workloads in each VM has been improved by 52.34% and 57.67% via using MemFlex. Since the execution

of the workloads triggers VM memory swapping, which is slow disk I/O operations in native system, however, MemFlex takes the advantages of dynamic shared memory to convert those disk I/O operations to memory copies, which significant mitigates the performance overhead introduced by VM memory swapping, thus improves the overall performance of the workloads running in the VMs. *Second*, we observe that the peak amount of swapped memory from each VM is 1.5GB, which means if the two VM swaps together, 3GB of shared memory are needed to accommodate all the swapped data. However, the peak swapping demands from the VMs do not appear at the same time even though the total size of shared memory for swapping is configured as 2GB. Therefore, the dynamic shared memory management mechanism is able to handle this overcommitment by flexibly moving the shared memory resources between the two VMs. It is important to note that the shared memory based VM memory swapping is only turned on when there is sufficient host memory available. MemFlex resorts to disk swapping when the total swapping area simultaneously demanded by the co-located VMs exceeds to size of the host shared memory region. *Third*, although memory intensive workloads, such as *h2* and *eclipse*, benefit most from shared memory swapping, MemFlex incurs negligible performance overhead for *non-memory intensive* workloads.

VI. Conclusion

Shared memory is widely used in virtualized cloud to mitigate the data accessing overhead and encouraging resource sharing among different VMs or between VMs and the host. However, current shared memory design is static and configured at compile time, which suffers from two inherent problems. First, the static management of shared memory cannot adaptive to the changing workloads and often leads to poor resource utilization under skewed or changing workloads. We have presented a dynamic shared memory management framework and how it can be deployed for improving inter-VM communication efficiency and VM memory swapping efficiency. Our experimental evaluation shows the effectiveness of our dynamic shared memory management framework.

Acknowledgment

This material is based upon work partially supported by the National Science Foundation under Grants IIS-0905493, CNS-1115375, IIP-1230740 and a grant from Intel ISTC on Cloud Computing.

References

[1] Netperf. *http://www.netperf.org/netperf/*.

[2] Spark. *https://spark.apache.org/*.

[3] Faraz Ahmad, Srimat T Chakradhar, Anand Raghunathan, and TN Vijaykumar. Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters. In *Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference*, pages 1–12. USENIX Association, 2014.

[4] Nadav Amit, Dan Tsafrir, and Assaf Schuster. Vswapper: A memory swapper for virtualized environments. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 349–366. ACM, 2014.

[5] Cristiana Amza, Alan L Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *Computer*, 29(2):18–28, 1996.

[6] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[7] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. The dacapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, volume 41, pages 169–190. ACM, 2006.

[8] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 133–146. ACM, 2009.

[9] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[10] Amit Gupta, Ehab Ababneh, Richard Han, and Eric Keller. Towards elastic operating systems. In *Proceedings of the 14th USENIX conference on Hot Topics in Operating Systems*, pages 16–16. USENIX Association, 2013.

[11] Wei Huang, Matthew J Koop, Qi Gao, and Dhabaleswar K Panda. Virtual machine aware communication libraries for high performance computing. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, page 9. ACM, 2007.

[12] Kangho Kim, Cheiyol Kim, Sung-In Jung, Hyun-Sup Shin, and Jin-Soo Kim. Inter-domain socket communications supporting high performance and full binary compatibility on xen. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 11–20. ACM, 2008.

[13] Prashanth Radhakrishnan and Kiran Srinivasan. Mmnet: An efficient inter-vm communication mechanism. *Proc. of Xen Summit. Boston*, 2008.

[14] M Wasi-ur Rahman, NS Islam, X Lu, J Jose, H Subramoni, H Wang, and DK Panda. High-performance rdma-based design of hadoop mapreduce over infiniband. In *Proceedings of the IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum. IPDPSW, Washington, DC, USA*, 2013.

[15] Yi Ren, Ling Liu, Qi Zhang, Qingbo Wu, Jie Wu, Jinzhu Kong, Jianbo Guan, and Huadong Dai. Residency-aware virtual machine communication optimization: Design choices and techniques. In *2013 IEEE 6th International Conference on Cloud Computing (CLOUD)*, pages 823–830. IEEE, 2013.

[16] Carl A Waldspurger. Memory resource management in vmware esx server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[17] Jian Wang, Kwame-Lante Wright, and Kartik Gopalan. Xenloop: a transparent high performance inter-vm network loopback. In *Proceedings of the 17th international symposium on High performance distributed computing*, pages 109–118. ACM, 2008.

[18] Dan Williams, Hani Jamjoom, Yew-Huey Liu, and Hakim Weatherspoon. Overdriver: Handling memory overload in an oversubscribed cloud. In *ACM SIGPLAN Notices*, volume 46, pages 205–216. ACM, 2011.

[19] Qi Zhang and Ling Liu. Workload adaptive shared memory management for high performance network i/o in virtualized cloud. In *Tech report*. Georgia Tech, 2015-4-17.

[20] Qi Zhang, Ling Liu, Gong Su, and Arun Iyengar. Memflex: Flexible and efficient memory resource management for high performance virtual machine execution. In *Tech report*. Georgia Tech and IBM, 2015-2-3.